Создание активных веб-страниц

6-е издание
Включает ECMAScript 5 и HTML5

JavaScript

Подробное руководство

Символ®
O'REILLY®

Дэвид Флэнаган

16 лет вместе
с профессионалами

с 1996 года
books.ru
всегда к слову

JavaScript

The Definitive Guide

Sixth Edition

David Flanagan

O'REILLY®

JavaScript

Подробное руководство

Шестое издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2012

Дэвид Флэнаган

JavaScript. Подробное руководство, 6-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*Ю. Бочина*
Корректор	*Т. Школьная*
Верстка	*Д. Орлова*

Флэнаган Д.

JavaScript. Подробное руководство, 6-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2012. – 1080 с., ил.

ISBN 978-5-93286-215-5

Шестое издание бестселлера «JavaScript. Подробное руководство» полностью пересмотрено и дополнено сведениями о JavaScript в соответствии с современным положением дел в разработке приложений для Web 2.0. Эта книга – одновременно и руководство программиста с большим числом практических примеров, и полноценный справочник по базовому языку JavaScript и клиентским прикладным интерфейсам, предоставляемым веб-броузерами.

Издание охватывает стандарты ECMAScript 5 и HTML5. Многие главы переписаны заново, другие дополнены новой информацией, появились и новые главы с описанием библиотеки jQuery и поддержки JavaScript на стороне сервера.

Часть I знакомит с основами JavaScript. В части II описывается среда разработки сценариев, предоставляемая веб-броузерами. Основное внимание уделяется разработке сценариев с применением методики ненавязчивого JavaScript и модели DOM. Часть III – обширный справочник по базовому языку JavaScript, включающий описания всех классов, объектов, конструкторов, методов, функций, свойств и констант, определенных в JavaScript 1.8, V8 3.0 и ECMAScript 5. Часть IV – справочник по клиентскому JavaScript. Здесь описываются API веб-броузеров, стандарт DOM API Level 3 и недавно вошедшие в стандарт HTML5 технологии WebSockets и WebWorkers, объекты localStorage и sessionStorage, а также теги <audio> и <video>.

Издание рекомендуется программистам, которым потребовалось изучить язык программирования для Веб, а также программистам, использующим язык JavaScript и желающим овладеть им в совершенстве.

ISBN 978-5-93286-215-5
ISBN 978-0-596-80552-4 (англ)

© Издательство Символ-Плюс, 2012
Authorized Russian translation of the English edition of JavaScript: The Definitive Guide, Sixth Edition ISBN 978-0-596-80552-4 © 2011 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Подписано в печать 30.05.2012. Формат 70×100$^1/_{16}$.
Печать офсетная. Объем 67,5 печ. л.

Эта книга посвящается всем,
кто учит жить в мире и противостоит насилию.

Оглавление

Предисловие 17

1. Введение в JavaScript 21
  1.1. Базовый JavaScript 25
  1.2. Клиентский JavaScript 29
    1.2.1. Пример: калькулятор платежей по ссуде на JavaScript 33

I. Базовый JavaScript 39

2. Лексическая структура 41
  2.1. Набор символов 41
    2.1.1. Чувствительность к регистру 41
    2.1.2. Пробелы, переводы строк и символы управления форматом 42
    2.1.3. Экранированные последовательности Юникода 42
    2.1.4. Нормализация 43
  2.2. Комментарии 43
  2.3. Литералы 43
  2.4. Идентификаторы и зарезервированные слова 44
    2.4.1. Зарезервированные слова 44
  2.5. Необязательные точки с запятой 46

3. Типы данных, значения и переменные 49
  3.1. Числа 51
    3.1.1. Целые литералы 52
    3.1.2. Литералы вещественных чисел 52
    3.1.3. Арифметические операции в JavaScript 53
    3.1.4. Двоичное представление вещественных чисел
    и ошибки округления 55
    3.1.5. Дата и время 56
  3.2. Текст 56
    3.2.1. Строковые литералы 56
    3.2.2. Управляющие последовательности в строковых литералах 58
    3.2.3. Работа со строками 59
    3.2.4. Сопоставление с шаблонами 60
  3.3. Логические значения 61
  3.4. Значения null и undefined 62
  3.5. Глобальный объект 63
  3.6. Объекты-обертки 64
  3.7. Неизменяемые простые значения и ссылки на изменяемые объекты 65

- 3.8. Преобразование типов 67
  - 3.8.1. Преобразования и равенство 68
  - 3.8.2. Явные преобразования 69
  - 3.8.3. Преобразование объектов в простые значения 71
- 3.9. Объявление переменных 74
  - 3.9.1 Повторные и опущенные объявления 74
- 3.10. Область видимости переменной 75
  - 3.10.1. Область видимости функции и подъем 76
  - 3.10.2. Переменные как свойства 77
  - 3.10.3 Цепочки областей видимости 78

4. Выражения и операторы 79

- 4.1. Первичные выражения 79
- 4.2. Инициализаторы объектов и массивов 80
- 4.3. Выражения определений функций 81
- 4.4. Выражения обращения к свойствам 82
- 4.5. Выражения вызова 83
- 4.6. Выражения создания объектов 84
- 4.7. Обзор операторов 84
  - 4.7.1. Количество операндов 86
  - 4.7.2. Типы данных операндов и результата 86
  - 4.7.3. Левосторонние выражения 86
  - 4.7.4. Побочные эффекты операторов 87
  - 4.7.5. Приоритет операторов 87
  - 4.7.6. Ассоциативность операторов 88
  - 4.7.7. Порядок вычисления 88
- 4.8. Арифметические выражения 88
  - 4.8.1. Оператор + ... 89
  - 4.8.2. Унарные арифметические операторы 90
  - 4.8.3. Поразрядные операторы 91
- 4.9. Выражения отношений 93
  - 4.9.1. Операторы равенства и неравенства 93
  - 4.9.2. Операторы сравнения 95
  - 4.9.3. Оператор in 97
  - 4.9.4. Оператор instanceof 97
- 4.10. Логические выражения 98
  - 4.10.1. Логическое И (&&) 98
  - 4.10.2. Логическое ИЛИ (||) 99
  - 4.10.3. Логическое НЕ (!) 100
- 4.11. Выражения присваивания 100
  - 4.11.1. Присваивание с операцией 101
- 4.12. Вычисление выражений 102
  - 4.12.1. eval() 103
  - 4.12.2. Использование eval() в глобальном контексте 103
  - 4.12.3. Использование eval() в строгом режиме 104
- 4.13. Прочие операторы 105
  - 4.13.1. Условный оператор (?:) 105
  - 4.13.2. Оператор typeof 105
  - 4.13.3. Оператор delete 107
  - 4.13.4. Оператор void 108
  - 4.13.5. Оператор «запятая» (,) 108

5. Инструкции 109
- 5.1. Инструкции-выражения 110
- 5.2. Составные и пустые инструкции 110
- 5.3. Инструкции-объявления 112
  - 5.3.1. Инструкция var 112
  - 5.3.2. Инструкция function 113
- 5.4. Условные инструкции 114
  - 5.4.1. Инструкция if 114
  - 5.4.2. Инструкция else if 116
  - 5.4.3. Инструкция switch 117
- 5.5. Циклы 119
  - 5.5.1. Инструкция while 119
  - 5.5.2. Инструкция do/while 120
  - 5.5.3. Инструкция for 121
  - 5.5.4. Инструкция for/in 122
- 5.6. Переходы 124
  - 5.6.1. Метки инструкций 124
  - 5.6.2. Инструкция break 125
  - 5.6.3. Инструкция continue 126
  - 5.6.4. Инструкция return 127
  - 5.6.5. Инструкция throw 128
  - 5.6.6. Инструкция try/catch/finally 129
- 5.7. Прочие инструкции 131
  - 5.7.1. Инструкция with 131
  - 5.7.2. Инструкция debugger 132
  - 5.7.3. "use strict" 133
- 5.8. Итоговая таблица JavaScript-инструкций 135

6. Объекты 137
- 6.1. Создание объектов 139
  - 6.1.1. Литералы объектов 139
  - 6.1.2. Создание объектов с помощью оператора new 140
  - 6.1.3. Прототипы 140
  - 6.1.4. Object.create() 141
- 6.2. Получение и изменение свойств 142
  - 6.2.1. Объекты как ассоциативные массивы 143
  - 6.2.2. Наследование 144
  - 6.2.3. Ошибки доступа к свойствам 145
- 6.3. Удаление свойств 147
- 6.4. Проверка существования свойств 148
- 6.5. Перечисление свойств 149
- 6.6. Методы чтения и записи свойств 152
- 6.7. Атрибуты свойств 154
  - 6.7.1. Устаревшие приемы работы с методами чтения и записи 157
- 6.8. Атрибуты объекта 158
  - 6.8.1. Атрибут prototype 158
  - 6.8.2. Атрибут class 159
  - 6.8.3. Атрибут extensible 160
- 6.9. Сериализация объектов 161
- 6.10. Методы класса Object 162
  - 6.10.1. Метод toString() 162
  - 6.10.2. Метод toLocaleString() 163

6.10.3. Метод toJSON() 163
6.10.4. Метод valueOf() 163

7. Массивы 164

7.1. Создание массивов 165
7.2. Чтение и запись элементов массива 166
7.3. Разреженные массивы 167
7.4. Длина массива 168
7.5. Добавление и удаление элементов массива 169
7.6. Обход элементов массива 170
7.7. Многомерные массивы 171
7.8. Методы класса Array 172
7.8.1. Метод join() 172
7.8.2. Метод reverse() 172
7.8.3. Метод sort() 173
7.8.4. Метод concat() 173
7.8.5. Метод slice() 174
7.8.6. Метод splice() 174
7.8.7. Методы push() и pop() 175
7.8.8. Методы unshift() и shift() 175
7.8.9. Методы toString() и toLocaleString() 176
7.9. Методы класса Array, определяемые стандартом ECMAScript 5 176
7.9.1. Метод forEach() 176
7.9.2. Метод map() 177
7.9.3. Метод filter() 177
7.9.4. Методы every() и some() 178
7.9.5. Методы reduce() и reduceRight() 178
7.9.6. Методы indexOf() и lastIndexOf() 180
7.10. Тип Array 181
7.11. Объекты, подобные массивам 182
7.12. Строки как массивы 184

8. Функции 185

8.1. Определение функций 185
8.1.1. Вложенные функции 188
8.2. Вызов функций 189
8.2.1. Вызов функций 189
8.2.2. Вызов методов 190
8.2.3. Вызов конструкторов 192
8.2.4. Косвенный вызов 193
8.3. Аргументы и параметры функций 193
8.3.1. Необязательные аргументы 193
8.3.2. Списки аргументов переменной длины: объект Arguments 194
8.3.3. Использование свойств объекта в качестве аргументов 196
8.3.4. Типы аргументов 197
8.4. Функции как данные 198
8.4.1. Определение собственных свойств функций 200
8.5. Функции как пространства имен 201
8.6. Замыкания 203
8.7. Свойства и методы функций и конструктор Function 209
8.7.1. Свойство length 209

8.7.2. Свойство prototype 209
8.7.3. Методы call() и apply() 210
8.7.4. Метод bind() 211
8.7.5. Метод toString() 213
8.7.6. Конструктор Function() 213
8.7.7. Вызываемые объекты 214
8.8. Функциональное программирование 215
8.8.1. Обработка массивов с помощью функций 215
8.8.2. Функции высшего порядка 217
8.8.3. Частичное применение функций 218
8.8.4. Мемоизация 220

9. Классы и модули 221
9.1. Классы и прототипы 222
9.2. Классы и конструкторы 223
9.2.1. Конструкторы и идентификация класса 225
9.2.2. Свойство constructor 226
9.3. Классы в стиле Java 227
9.4. Наращивание возможностей классов 231
9.5. Классы и типы 232
9.5.1. Оператор instanceof 232
9.5.2. Свойство constructor 233
9.5.3. Имя конструктора 234
9.5.4. Грубое определение типа 235
9.6. Приемы объектно-ориентированного программирования в JavaScript 238
9.6.1. Пример: класс множества 238
9.6.2. Пример: типы-перечисления 239
9.6.3. Стандартные методы преобразований 242
9.6.4. Методы сравнения 244
9.6.5. Заимствование методов 247
9.6.6. Частные члены 249
9.6.7. Перегрузка конструкторов и фабричные методы 250
9.7. Подклассы 252
9.7.1. Определение подкласса 252
9.7.2. Вызов конструктора и методов базового класса 254
9.7.3. Композиция в сравнении с наследованием 256
9.7.4. Иерархии классов и абстрактные классы 258
9.8. Классы в ECMAScript 5 262
9.8.1. Определение неперечислимых свойств 262
9.8.2. Определение неизменяемых классов 263
9.8.3. Сокрытие данных объекта 265
9.8.4. Предотвращение расширения класса 266
9.8.5. Подклассы и ECMAScript 5 267
9.8.6. Дескрипторы свойств 268
9.9. Модули 270
9.9.1. Объекты как пространства имен 271
9.9.2. Область видимости функции как частное пространство имен 273

10. Шаблоны и регулярные выражения 276
10.1. Определение регулярных выражений 276
10.1.1. Символы литералов 277
10.1.2. Классы символов 278

10.1.3. Повторение . . . 279
10.1.4. Альтернативы, группировка и ссылки . . . 281
10.1.5. Указание позиции соответствия . . . 283
10.1.6. Флаги . . . 284
10.2. Методы класса String для поиска по шаблону . . . 285
10.3. Объект RegExp . . . 287
10.3.1. Свойства RegExp . . . 288
10.3.2. Методы RegExp . . . 288

11. Подмножества и расширения JavaScript . . . 290
11.1. Подмножества JavaScript . . . 291
11.1.1. Подмножество The Good Parts . . . 291
11.1.2. Безопасные подмножества . . . 292
11.2. Константы и контекстные переменные . . . 295
11.3. Присваивание с разложением . . . 298
11.4. Итерации . . . 300
11.4.1. Цикл for/each . . . 300
11.4.2. Итераторы . . . 301
11.4.3. Генераторы . . . 303
11.4.4. Генераторы массивов . . . 307
11.4.5. Выражения-генераторы . . . 308
11.5. Краткая форма записи функций . . . 309
11.6. Множественные блоки catch . . . 309
11.7. E4X: ECMAScript for XML . . . 310

12. Серверный JavaScript . . . 314
12.1. Управление Java с помощью Rhino . . . 315
12.1.1. Пример использования Rhino . . . 319
12.2. Асинхронный ввод/вывод в интерпретаторе Node . . . 321
12.2.1. Пример использования Node: HTTP-сервер . . . 327
12.2.2. Пример использования Node: модуль утилит клиента HTTP . . . 329

II. Клиентский JavaScript . . . 331

13. JavaScript в веб-броузерах . . . 333
13.1. Клиентский JavaScript . . . 333
13.1.1. Сценарии JavaScript в веб-документах . . . 336
13.1.2. Сценарии JavaScript в веб-приложениях . . . 336
13.2. Встраивание JavaScript-кода в разметку HTML . . . 337
13.2.1. Элемент <script> . . . 338
13.2.2. Сценарии во внешних файлах . . . 339
13.2.3. Тип сценария . . . 340
13.2.4. Обработчики событий в HTML . . . 341
13.2.5. JavaScript в URL . . . 342
13.3. Выполнение JavaScript-программ . . . 344
13.3.1. Синхронные, асинхронные и отложенные сценарии . . . 345
13.3.2. Выполнение, управляемое событиями . . . 347
13.3.3. Модель потоков выполнения в клиентском JavaScript . . . 349
13.3.4. Последовательность выполнения клиентских сценариев . . . 350
13.4. Совместимость на стороне клиента . . . 352
13.4.1. Библиотеки обеспечения совместимости . . . 356

13.4.2. Классификация броузеров 356
13.4.3. Проверка особенностей 357
13.4.4. Режим совместимости и стандартный режим 358
13.4.5. Проверка типа броузера 358
13.4.6. Условные комментарии в Internet Explorer 359
13.5. Доступность 360
13.6. Безопасность 361
13.6.1. Чего не может JavaScript 362
13.6.2. Политика общего происхождения 363
13.6.3. Взаимодействие с модулями расширения и элементами управления ActiveX 365
13.6.4. Межсайтовый скриптинг 365
13.6.5. Атаки типа отказа в обслуживании 367
13.7. Клиентские фреймворки 367

14. Объект Window 369
14.1. Таймеры 370
14.2. Адрес документа и навигация по нему 371
14.2.1. Анализ URL 371
14.2.2. Загрузка нового документа 372
14.3. История посещений 373
14.4. Информация о броузере и об экране 374
14.4.1. Объект Navigator 374
14.4.2. Объект Screen 377
14.5. Диалоги 377
14.6. Обработка ошибок 379
14.7. Элементы документа как свойства окна 380
14.8. Работа с несколькими окнами и фреймами 382
14.8.1. Открытие и закрытие окон 383
14.8.2. Отношения между фреймами 385
14.8.3. JavaScript во взаимодействующих окнах 387

15. Работа с документами 390
15.1. Обзор модели DOM 390
15.2. Выбор элементов документа 393
15.2.1. Выбор элементов по значению атрибута id 393
15.2.2. Выбор элементов по значению атрибута name 394
15.2.3. Выбор элементов по типу 395
15.2.4. Выбор элементов по классу CSS 397
15.2.5. Выбор элементов с использованием селекторов CSS 398
15.2.6. document.all[] 400
15.3. Структура документа и навигация по документу 401
15.3.1. Документы как деревья узлов 401
15.3.2. Документы как деревья элементов 402
15.4. Атрибуты 405
15.4.1. HTML-атрибуты как свойства объектов Element 405
15.4.2. Доступ к нестандартным HTML-атрибутам 406
15.4.3. Атрибуты с данными 407
15.4.4. Атрибуты как узлы типа Attr 408
15.5. Содержимое элемента 409
15.5.1. Содержимое элемента в виде HTML 409
15.5.2. Содержимое элемента в виде простого текста 410

15.5.3. Содержимое элемента в виде текстовых узлов 411
15.6. Создание, вставка и удаление узлов 412
15.6.1. Создание узлов 413
15.6.2. Вставка узлов 413
15.6.3. Удаление и замена узлов 415
15.6.4. Использование объектов DocumentFragment 416
15.7. Пример: создание оглавления 418
15.8. Геометрия документа и элементов и прокрутка 421
15.8.1. Координаты документа и видимой области 421
15.8.2. Определение геометрии элемента 423
15.8.3. Определение элемента в указанной точке 424
15.8.4. Прокрутка 425
15.8.5. Подробнее о размерах, позициях и переполнении элементов 426
15.9. HTML-формы 428
15.9.1. Выбор форм и элементов форм 430
15.9.2. Свойства форм и их элементов 431
15.9.3. Обработчики событий форм и их элементов 432
15.9.4. Кнопки 433
15.9.5. Переключатели и флажки 434
15.9.6. Текстовые поля ввода 434
15.9.7. Элементы Select и Option 435
15.10. Другие особенности документов 437
15.10.1. Свойства объекта Document 437
15.10.2. Метод document.write() 438
15.10.3. Получение выделенного текста 440
15.10.4. Редактируемое содержимое 441

16. Каскадные таблицы стилей 444
16.1. Обзор CSS 445
16.1.1. Каскад правил 446
16.1.2. История развития CSS 447
16.1.3. Сокращенная форма определения свойств 447
16.1.4. Нестандартные свойства 447
16.1.5. Пример CSS-таблицы 448
16.2. Наиболее важные CSS-свойства 450
16.2.1. Позиционирование элементов с помощью CSS 451
16.2.2. Рамки, поля и отступы 454
16.2.3. Блочная модель и детали позиционирования в CSS 455
16.2.4. Отображение и видимость элементов 457
16.2.5. Цвет, прозрачность и полупрозрачность 458
16.2.6. Частичная видимость: свойства overflow и clip 459
16.2.7. Пример: перекрытие полупрозрачных окон 460
16.3. Управление встроенными стилями 463
16.3.1. Создание анимационных эффектов средствами CSS 465
16.4. Вычисленные стили 468
16.5. CSS-классы 470
16.6. Управление таблицами стилей 472
16.6.1. Включение и выключение таблиц стилей 472
16.6.2. Получение, вставка и удаление правил из таблиц стилей 473
16.6.3. Создание новых таблиц стилей 474

17. Обработка событий 476
- 17.1. Типы событий 479
  - 17.1.1. Старые типы событий 479
  - 17.1.2. События модели DOM 485
  - 17.1.3. События HTML5 486
  - 17.1.4. События, генерируемые сенсорными экранами и мобильными устройствами 488
- 17.2. Регистрация обработчиков событий 489
  - 17.2.1. Установка свойств обработчиков событий 489
  - 17.2.2. Установка атрибутов обработчиков событий 490
  - 17.2.3. addEventListener() 491
  - 17.2.4. attachEvent() 492
- 17.3. Вызов обработчиков событий 492
  - 17.3.1. Аргумент обработчика событий 493
  - 17.3.2. Контекст обработчиков событий 493
  - 17.3.3. Область видимости обработчика событий 494
  - 17.3.4. Возвращаемые значения обработчиков 495
  - 17.3.5. Порядок вызова 495
  - 17.3.6. Распространение событий 496
  - 17.3.7. Отмена событий 497
- 17.4. События загрузки документа 498
- 17.5. События мыши 500
- 17.6. События колесика мыши 504
- 17.7. События механизма буксировки (drag-and-drop) 508
- 17.8. События ввода текста 515
- 17.9. События клавиатуры 518

18. Работа с протоколом HTTP 524
- 18.1. Использование объекта XMLHttpRequest 527
  - 18.1.1. Выполнение запроса 529
  - 18.1.2. Получение ответа 531
  - 18.1.3. Оформление тела запроса 535
  - 18.1.4. События, возникающие в ходе выполнения HTTP-запроса 541
  - 18.1.5. Прерывание запросов и предельное время ожидания 544
  - 18.1.6. Выполнение междоменных HTTP-запросов 545
- 18.2. Выполнение HTTP-запросов с помощью <script>: JSONP 548
- 18.3. Архитектура Comet на основе стандарта «Server-Sent Events» 550

19. Библиотека jQuery 556
- 19.1. Основы jQuery 557
  - 19.1.1. Функция jQuery() 558
  - 19.1.2. Запросы и результаты запросов 562
- 19.2. Методы чтения и записи объекта jQuery 565
  - 19.2.1. Чтение и запись значений HTML-атрибутов 565
  - 19.2.2. Чтение и запись значений CSS-атрибутов 566
  - 19.2.3. Чтение и запись CSS-классов 566
  - 19.2.4. Чтение и запись значений элементов HTML-форм 567
  - 19.2.5. Чтение и запись содержимого элемента 568
  - 19.2.6. Чтение и запись параметров геометрии элемента 568
  - 19.2.7. Чтение и запись данных в элементе 570
- 19.3. Изменение структуры документа 571

- 19.3.1. Вставка и замена элементов 571
- 19.3.2. Копирование элементов 573
- 19.3.3. Обертывание элементов 574
- 19.3.4. Удаление элементов 574
- 19.4. Обработка событий с помощью библиотеки jQuery 575
  - 19.4.1. Простые методы регистрации обработчиков событий 575
  - 19.4.2. Обработчики событий в библиотеке jQuery 577
  - 19.4.3. Объект Event в библиотеке jQuery 577
  - 19.4.4. Дополнительные способы регистрации обработчиков событий 579
  - 19.4.5. Удаление обработчиков событий 580
  - 19.4.6. Возбуждение событий 582
  - 19.4.7. Реализация собственных событий 584
  - 19.4.8. Динамические события 584
- 19.5. Анимационные эффекты 586
  - 19.5.1. Простые эффекты 588
  - 19.5.2. Реализация собственных анимационных эффектов 589
  - 19.5.3. Отмена, задержка и постановка эффектов в очередь 593
- 19.6. Реализация Ajax в библиотеке jQuery 595
  - 19.6.1. Метод load() 595
  - 19.6.2. Вспомогательные функции поддержки Ajax 597
  - 19.6.3. Функция jQuery.ajax() 602
  - 19.6.4. События в архитектуре Ajax 608
- 19.7. Вспомогательные функции 610
- 19.8. Селекторы и методы выбора в библиотеке jQuery 613
  - 19.8.1. Селекторы jQuery 613
  - 19.8.2. Методы выбора 618
- 19.9. Расширение библиотеки jQuery с помощью модулей расширений 622
- 19.10. Библиотека jQuery UI 625

20. Сохранение данных на стороне клиента 627

- 20.1. Объекты localStorage и sessionStorage 630
  - 20.1.1. Срок хранения и область видимости 630
  - 20.1.2. Прикладной программный интерфейс объекта Storage 632
  - 20.1.3. События объекта Storage 633
- 20.2. Cookies 634
  - 20.2.1. Атрибуты cookie: срок хранения и область видимости 635
  - 20.2.2. Сохранение cookies 637
  - 20.2.3. Чтение cookies 638
  - 20.2.4. Ограничения cookies 639
  - 20.2.5. Реализация хранилища на основе cookies 639
- 20.3. Механизм сохранения userData в IE 641
- 20.4. Хранилище приложений и автономные веб-приложения 643
  - 20.4.1. Объявление кэшируемого приложения 643
  - 20.4.2. Обновление кэша 645
  - 20.4.3. Автономные веб-приложения 650

21. Работа с графикой и медиафайлами на стороне клиента 655

- 21.1. Работа с готовыми изображениями 656
  - 21.1.1. Ненавязчивая реализация смены изображений 657
- 21.2. Работа с аудио- и видеопотоками 658
  - 21.2.1. Выбор типа и загрузка 659
  - 21.2.2. Управление воспроизведением 660

21.2.3. Определение состояния мультимедийных элементов . . . 661
21.2.4 События мультимедийных элементов . . . 663
21.3. SVG – масштабируемая векторная графика . . . 665
21.4. Создание графики с помощью элемента <canvas> . . . 672
21.4.1. Рисование линий и заливка многоугольников . . . 675
21.4.2. Графические атрибуты . . . 678
21.4.3. Размеры и система координат холста . . . 679
21.4.4. Преобразование системы координат . . . 680
21.4.5. Рисование и заливка кривых . . . 685
21.4.6. Прямоугольники . . . 687
21.4.7. Цвет, прозрачность, градиенты и шаблоны . . . 687
21.4.8. Атрибуты рисования линий . . . 691
21.4.9. Текст . . . 692
21.4.10. Отсечение . . . 693
21.4.11. Тени . . . 695
21.4.12. Изображения . . . 696
21.4.13. Композиция . . . 698
21.4.14. Манипулирование пикселами . . . 701
21.4.15. Определение попадания . . . 703
21.4.16. Пример использования элемента <canvas>: внутристрочные диаграммы . . . 704

22. Прикладные интерфейсы HTML5 . . . 706
22.1. Геопозиционирование . . . 707
22.2. Управление историей посещений . . . 711
22.3. Взаимодействие документов с разным происхождением . . . 716
22.4. Фоновые потоки выполнения . . . 720
22.4.1. Объект Worker . . . 721
22.4.2. Область видимости фонового потока . . . 722
22.4.3. Примеры использования фоновых потоков . . . 725
22.5. Типизированные массивы и буферы . . . 728
22.6. Двоичные объекты . . . 732
22.6.1. Файлы как двоичные объекты . . . 734
22.6.2. Загрузка двоичных объектов . . . 735
22.6.3. Конструирование двоичных объектов . . . 736
22.6.4. URL-адреса, ссылающиеся на двоичные объекты . . . 736
22.6.5. Чтение двоичных объектов . . . 739
22.7. Прикладной интерфейс к файловой системе . . . 742
22.8. Базы данных на стороне клиента . . . 747
22.9. Веб-сокеты . . . 755

III. Справочник по базовому JavaScript . . . 759

Справочник по базовому JavaScript . . . 761

IV. Справочник по клиентскому JavaScript . . . 881

Справочник по клиентскому JavaScript . . . 883

Алфавитный указатель . . . 1040

Предисловие

Эта книга охватывает язык программирования JavaScript и прикладные интерфейсы JavaScript, реализованные в веб-броузерах. Я писал ее для тех, кто уже имеет некоторый опыт программирования и желает изучить JavaScript, а также для программистов, уже использующих JavaScript, но стремящихся подняться на более высокий уровень мастерства и по-настоящему овладеть языком и веб-платформой. Моя цель состояла в том, чтобы максимально полно и подробно описать JavaScript и платформу. В результате получилась эта объемная и подробная книга. Однако смею надеяться, что вы будете вознаграждены за внимательное изучение книги и время, потраченное на ее чтение, будет компенсировано более высокой производительностью труда.

Книга делится на четыре части. Часть I охватывает сам язык JavaScript. Часть II охватывает клиентский JavaScript: прикладные программные интерфейсы JavaScript, определяемые стандартом HTML5 и сопутствующими ему стандартами и реализованные в веб-броузерах. Часть III представляет собой справочник по базовому языку, а часть IV – справочник по клиентскому JavaScript. Глава 1 включает краткий обзор глав первой и второй частей книги (раздел 1.1).

Это шестое издание книги охватывает стандарты ECMAScript 5 (последняя версия спецификации базового языка) и HTML5 (последняя версия спецификации веб-платформы). Положения стандарта ECMAScript 5 будут рассматриваться на протяжении всей первой части. Нововведения, появившиеся в HTML5, в основном будут обсуждаться в конце части II, но мы будем рассматривать их и в других главах. Совершенно новыми в этом издании являются глава 11 «Подмножества и расширения JavaScript», глава 12 «Серверный JavaScript», глава 19 «Библиотека jQuery» и глава 22 «Прикладные интерфейсы HTML5».

Читатели предыдущих изданий могут заметить, что в этом издании я полностью переписал многие главы. Главы первой части книги, посвященные основам языка и охватывающие объекты, массивы, функции и классы, были переписаны заново и приведены в соответствие с современными приемами программирования. Ключевые главы второй части, описывающие документы и события, точно так же были полностью переписаны, чтобы привести их к современному уровню.

Несколько слов о пиратстве

Если вы читаете электронную версию этой книги, за которую вы (или ваш работодатель) ничего не платили (или позаимствовали ее у третьего лица, не заплатившего за книгу), то, скорее всего, эта копия является пиратской. Работа над шестым изданием продолжалась более года, и мне приходилось трудиться над книгой полный рабочий день в течение всего этого времени. Оплату за свой труд я получаю, лишь когда кто-то покупает эту книгу. И единственным источником дохода, который позволит мне продолжить работу над седьмым изданием, является гонорар от продажи шестого издания.

Я не приветствую пиратство, но, если вы читаете пиратскую копию, прочитайте несколько глав. Это позволит вам убедиться, что данная книга является ценным источником информации о JavaScript, лучше организованным и более качественным, чем бесплатные (и законные) источники информации, доступные в Веб. Если вы согласитесь с тем, что эта книга является ценным источником информации, пожалуйста, заплатите за эту ценность, приобретя легальную копию книги (электронную или бумажную). Если же вы посчитаете, что эта книга ничуть не лучше открытых источников информации в Веб, пожалуйста, уничтожьте пиратскую копию и пользуйтесь открытыми источниками информации.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Обозначает первое вхождение термина. *Курсив* также применяется для выделения адресов электронной почты, адресов URL и имен файлов и каталогов.

`Моноширинный шрифт`

Применяется для форматирования программного кода на языке JavaScript, листингов CSS и HTML и вообще всего, что непосредственно набирается на клавиатуре при программировании.

`Моноширинный курсив`

Обозначает аргументы функций и любые другие элементы, которые в программе необходимо заменить реальными значениями.

Использование программного кода примеров

Примеры для этой книги доступны в электронном виде. Соответствующие ссылки можно найти на странице книги на веб-сайте издательства:

http://oreilly.com/catalog/9780596805531/

Данная книга призвана оказать помощь в решении ваших задач. Вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и задействуете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам *необходимо* получить разрешение от издательства O'Reilly. При цитировании данной книги или примеров из нее и при ответе на вопросы получение разрешения не требуется. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам *необходимо* получить разрешение издательства.

Если вы соберетесь использовать программный код из этой книги, я приветствую, но не требую добавлять ссылку на первоисточник при цитировании. Под ссылкой подразумевается указание авторов, издательства и ISBN. Например: «JavaScript: The Definitive Guide, by David Flanagan (O'Reilly). Copyright 2011 David Flanagan, 978-0-596-80552-4».

Дополнительную информацию о порядке использования программного кода примеров можно найти на странице *http://oreilly.com/pub/a/oreilly/ask_tim/2001/codepolicy.html*. За получением разрешения на использование значительных объмов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Ошибки и контактная информация

Издательство на своем сайте публикует список ошибок, найденных в книге. Вы можете ознакомиться со списком и отправить свои замечания об обнаруженных вами ошибках, посетив веб-страницу:

http://oreilly.com/catalog/9780596805531

С вопросами и предложениями технического характера, касающимися этой книги, обращайтесь по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на веб-сайте:

http://www.oreilly.com

Можно найти нас на сайте Facebook: *http://facebook.com/oreilly*

Следить за последними новостями на сайте Twitter:

http://twitter.com/oreillymedia.

Просматривать видеоматериалы на сайте YouTube:

http://www.youtube.com/oreillymedia.

Благодарности

Работая над этой книгой, я получал помощь от многих людей. Я хотел бы поблагодарить моего редактора, Майка Лукидеса (Mike Loukides), который старался удержать меня в рамках планов и вносил полезные комментарии. Спасибо также моим техническим редакторам: Захару Кессину (Zachary Kessin), просмотревшему многие главы в первой части, и Рафаэлю Цекко (Raffaele Cecco), который редактировал главу 19 и материал с описанием тега `<canvas>` в главе 21. Как обычно, блестяще справился со своей работой производственный отдел издательства O'Reilly: Дэн Фокссмит (Dan Fauxsmith), руководивший производственным процессом, Тереза Элси (Teresa Elsey), выполнявшая литературную правку, Роб Романо (Rob Romano), готовивший иллюстрации, и Элен Тротман Цайг (Ellen Troutman Zaig), создавшая алфавитный указатель.

В нашу эпоху широкого развития средств электронной коммуникации практически невозможно перечислить всех, кто оказывал помощь в том или ином виде. И мне хотелось бы поблагодарить всех, кто отвечал на мои вопросы, касающиеся ECMAScript 5, w3c, всех участников списков рассылки и всех, кто делился со мной своими идеями, касающимися программирования на JavaScript. Я глубоко сожалею, что не могу перечислить всех поименно, но хочу сказать, что мне было приятно работать с таким ярким сообществом программистов на JavaScript.

В работе над предыдущими изданиями книги принимали участие следующие редакторы и рецензенты: Эндрю Шульман (Andrew Schulman), Анжело Сиригос (Angelo Sirigos), Аристотель Пагальцис (Aristotle Pagaltzis), Брендан Эйх (Brendan Eich), Кристиан Хейльманн (Christian Heilmann), Дэн Шейфер (Dan Shafer), Дэйв С. Митчелл (Dave C. Mitchell), Деб Камерон (Deb Cameron), Дуглас Крокфорд (Douglas Crockford), д-р Танкред Хиршманн (Dr. Tankred Hirschmann), Дилан Шиман (Dylan Schiemann), Френк Уиллисон (Frank Willison), Джефф Штернс (Geoff Stearns), Герман Вентер (Herman Venter), Джей Ходжес (Jay Hodges), Джефф Ятс (Jeff Yates), Джозеф Кесселман (Joseph Kesselman), Кен Купер (Ken Cooper), Ларри Салливан (Larry Sullivan), Линн Роллинс (Lynn Rollins), Нил Беркман (Neil Berkman), Ник Томпсон (Nick Thompson), Норрис Бойд (Norris Boyd), Паула Фергюсон (Paula Ferguson), Питер-Пауль Кох (Peter-Paul Koch), Филипп Ле Хегарет (Philippe Le Hegaret), Ричард Якер (Richard Yaker), Сандерс Клейнфельд (Sanders Kleinfeld), Скотт Фурман (Scott Furman), Скотт Иссакс (Scott Issacs), Шон Каценбергер (Shon Katzenberger), Терри Аллен (Terry Allen), Тодд Дихендорф (Todd Ditchendorf), Вайдур Аппарао (Vidur Apparao) и Валдемар Хорват (Waldemar Horwat).

Это издание книги было в значительной степени переписано заново, из-за чего моя семья провела массу вечеров без меня. Хочу выразить им мою любовь и благодарность за то, что терпели мое отсутствие.

Дэвид Флэнаган (*http://www.davidflanagan.com*), март 2011

1

Введение в JavaScript

JavaScript – это язык программирования для Веб. Подавляющее большинство веб-сайтов используют JavaScript, и все современные веб-броузеры – для настольных компьютеров, игровых приставок, электронных планшетов и смартфонов – включают интерпретатор JavaScript, что делает JavaScript самым широкоприменимым языком программирования из когда-либо существовавших в истории. JavaScript входит в тройку технологий, которые должен знать любой веб-разработчик: язык разметки HTML, позволяющий определять содержимое веб-страниц, язык стилей CSS, позволяющий определять внешний вид веб-страниц, и язык программирования JavaScript, позволяющий определять поведение веб-страниц. Эта книга поможет вам овладеть языком программирования.

Если вы знаете другие языки программирования, вам может оказаться полезна информация, что JavaScript является высокоуровневым, динамическим, нетипизированным и интерпретируемым языком программирования, который хорошо подходит для программирования в объектно-ориентированном и функциональном стилях. Свой синтаксис JavaScript унаследовал из языка Java, свои первоклассные функции – из языка Scheme, а механизм наследования на основе прототипов – из языка Self. Но вам не требуется знать все эти языки или быть знакомыми с их терминологией для чтения этой книги и изучения JavaScript.

Название языка «JavaScript» может вводить в заблуждение. За исключением поверхностной синтаксической схожести, JavaScript полностью отличается от языка программирования Java. JavaScript давно перерос рамки языка сценариев, превратившись в надежный и эффективный универсальный язык программирования. Последняя версия языка (смотрите врезку) определяет множество новых особенностей, позволяющих использовать его для разработки крупномасштабного программного обеспечения.

JavaScript: названия и версии

JavaScript был создан в компании Netscape на заре зарождения Веб. Название «JavaScript» является торговой маркой, зарегистрированной компанией Sun Microsystems (ныне Oracle), и используется для обозначения реализации языка, созданной компанией Netscape (ныне Mozilla). Компания Netscape представила язык для стандартизации европейской ассоциации производителей компьютеров ECMA (European Computer Manufacturer's Association), но из-за юридических проблем с торговыми марками стандартизованная версия языка получила несколько неуклюжее название «ECMAScript». Из-за тех же юридических проблем версия языка от компании Microsoft получила официальное название «JScript». Однако на практике все эти реализации обычно называют JavaScript. В этой книге мы будем использовать название «ECMAScript» только для ссылки на стандарт языка.

В течение прошлого десятилетия все веб-броузеры предоставляли реализацию версии 3 стандарта ECMAScript, и в действительности разработчикам не было необходимости задумываться о номерах версий: стандарт языка был стабилен, а его реализации в веб-броузерах в значительной мере были совместимыми. Недавно вышла новая важная версия стандарта языка под названием ECMAScript 5, и к моменту написания этих строк производители броузеров приступили к созданию его реализации. Эта книга охватывает все нововведения, появившиеся в ECMAScript 5, а также все особенности, предусмотренные стандартом ECMAScript 3. Иногда вам будут встречаться названия этих версий языка, сокращенные до ES3 и ES5, а также название JavaScript, сокращенное до JS.

Когда речь заходит непосредственно о самом языке, при этом подразумеваются только версии 3 и 5 стандарта ECMAScript. (Четвертая версия стандарта ECMAScript разрабатывалась много лет, но из-за слишком амбициозных целей так и не была выпущена.) Однако иногда можно встретить упоминание о версии JavaScript, например: JavaScript 1.5 или JavaScript 1.8. Эти номера версий присваивались реализациям JavaScript, выпускаемым компанией Mozilla, причем версия 1.5 соответствует базовому стандарту ECMAScript 3, а более высокие версии включают нестандартные расширения (подробнее об этом рассказывается в главе 11). Наконец, номера версий также присваиваются отдельным интерпретаторам, или «механизмам» JavaScript. Например, компания Google разрабатывает свой интерпретатор JavaScript под названием V8, и к моменту написания этих строк текущей версией механизма V8 была версия 3.0.

Чтобы представлять хоть какой-то интерес, каждый язык программирования должен иметь свою платформу, или стандартную библиотеку, или API функций для выполнения таких базовых операций, как ввод и вывод. Ядро языка JavaScript определяет минимальный прикладной интерфейс для работы с текстом, массивами, датами и регулярными выражениями, но в нем отсутствуют операции ввода-вывода. Ввод и вывод (а также более сложные возможности, такие как сетевые взаимодействия, сохранение данных и работа с графикой) перекладываются на «окружающую среду», куда встраивается JavaScript. Обычно роль окружающей среды играет веб-броузер (однако в главе 12 мы увидим два примера использования JavaScript без привлечения веб-броузера). Первая часть этой книги охватывает сам язык JavaScript и его минимальный прикладной интерфейс. Вторая часть описывает использование JavaScript в веб-броузерах и охватывает прикладной интерфейс, предоставляемый броузерами, который иногда называют «клиентским JavaScript».

Третья часть книги представляет собой справочник по базовому API языка. Например, чтобы ознакомиться с прикладным интерфейсом JavaScript для работы с массивами, вы можете отыскать и прочитать раздел «Array» в этой части книги. Четвертая часть – это справочник по клиентскому JavaScript. Например, чтобы ознакомиться с прикладным интерфейсом JavaScript для работы с графикой, определяемым стандартом HTML5 для тега `<canvas>`, можно отыскать и прочитать раздел «Canvas» в четвертой части книги.

В этой книге мы сначала рассмотрим низкоуровневые основы, а затем перейдем к базирущимся на них высокоуровневым абстракциям. Желательно читать главы, придерживаясь порядка, в котором они следуют в книге. Однако изучение нового языка программирования никогда не было линейным процессом, точно так же и описание языка трудно представить в линейном виде: каждая особенность языка тесно связана с другими особенностями, поэтому данная книга полна перекрестных ссылок – иногда назад, а иногда вперед – на сведения, с которыми вы еще не ознакомились. Эта глава являет собой первый краткий обзор основ языка и прикладного интерфейса клиентского JavaScript и представляет ключевые особенности, чем упрощает более глубокое их изучение в последующих главах.

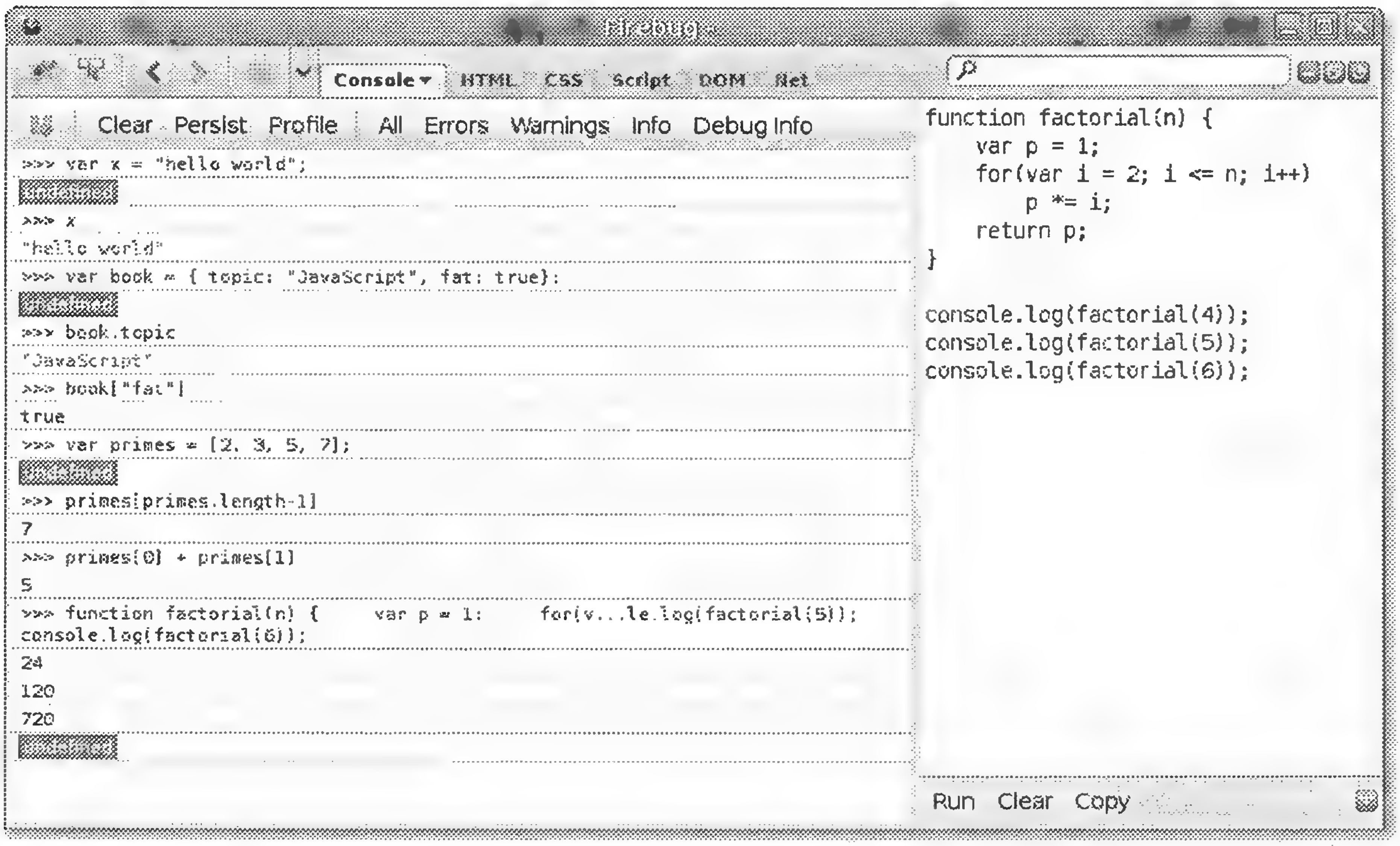
Исследование JavaScript

Изучая новый язык программирования, очень важно стараться пробовать запускать примеры, представленные в книге, изменять их и опять запускать, чтобы проверить, насколько правильно вы понимаете особенности языка. Для этого необходим интерпретатор JavaScript. К счастью, любой веб-броузер включает интерпретатор JavaScript, а если вы читаете эту книгу, у вас, скорее всего, на компьютере установлено более одного веб-броузера.

Далее в этой главе мы увидим, что код на языке JavaScript можно встраивать в HTML-файлы, в теги `<script>`, и при загрузке HTML-файла этот код будет выполняться броузером. К счастью, нам не требуется поступать так всякий раз, когда нужно опробовать короткий фрагмент программного кода JavaScript. Появление мощного и оригинального расширения Firebug для Firefox (изображено на рис. 1.1 и доступно для загрузки на сайте *http://getfirebug.com/*) подтолкнуло производителей веб-броузеров к включению в них инструментов веб-разработчика, необходимых для отладки, проверки и изучения. Обычно эти инструменты можно отыскать в меню Tools (Инструменты или Сервис) броузера в виде пункта Developer Tools (Средства разработчика) или Web Console (Веб-консоль). (Броузер Firefox 4 включает собственный встроенный инструмент Web Console, но к настоящему моменту расширение Firebug обладает более широкими возможностями.) Как правило, консоль можно запустить нажатием горячей комбинации клавиш, такой как F12 или Ctrl-Shift-J. Обычно эти инструменты открываются в виде отдельной панели в верхней или нижней части окна броузера, но некоторые броузеры открывают их в отдельном окне (как изображено на рис. 1.1), что часто более удобно.

Панель или окно типичного «инструмента разработчика» включает множество вкладок, позволяющих исследовать структуру HTML-документа, стили CSS, наблюдать за выполнением сетевых запросов и т. д. Среди них имеется вкладка JavaScript console (Консоль JavaScript), где можно вводить строки программного кода JavaScript и выполнять их. Это самый простой способ поэкспериментировать с JavaScript, и я рекомендую использовать его во время чтения этой книги.

В современных броузерах имеется простой переносимый API консоли. Для вывода текста в консоль можно использовать функцию `console.log()`. Зачастую такая возможность оказывается удивительно полезной при отладке, и некоторые примеры из этой книги (даже в разделе, посвященном базовому языку) используют `console.log()` для вывода простого текста. Похожий, но более навязчивый способ вывода информации или отладочных сообщений заключается в передаче строки текста функции `alert()`, которая отображает его в окне модального диалога.



```
>>> var x = "hello world";
>>> x
"hello world"
>>> var book = { topic: "JavaScript", fat: true};
>>> book.topic
"JavaScript"
>>> book["fat"]
true
>>> var primes = [2, 3, 5, 7];
>>> primes[primes.length-1]
7
>>> primes[0] + primes[1]
5
>>> function factorial(n) {     var p = 1;     for(v...le.log(factorial(5));
console.log(factorial(6));
24
120
720
```

```
function factorial(n) {
    var p = 1;
    for(var i = 2; i <= n; i++)
        p *= i;
    return p;
}

console.log(factorial(4));
console.log(factorial(5));
console.log(factorial(6));
```

Рис. 1.1. Отладочная консоль расширения Firebug для Firefox

1.1. Базовый JavaScript

Этот раздел представляет собой обзор языка JavaScript, а также обзор первой части этой книги. После этой вводной главы мы опустимся на самый нижний уровень JavaScript: в главе 2 «Лексическая структура» будут описаны основные лексические конструкции JavaScript, такие как комментарии, точки с запятой и набор символов Юникода. В главе 3 «Типы данных, значения и переменные» мы начнем рассматривать более интересные темы: здесь будут описаны переменные JavaScript и значения, которые можно присваивать этим переменным. Ниже приводится пример программного кода, иллюстрирующий предмет обсуждения этих двух глав:

```
// Все, что следует за двумя символами слэша, является комментарием.
// Внимательно читайте комментарии: они описывают программный код JavaScript.

// Переменная - это символическое имя некоторого значения.
// Переменные объявляются с помощью ключевого слова var:
var x;          // Объявление переменной с именем x.

// Присваивать значения переменным можно с помощью знака =
x = 0;          // Теперь переменная x имеет значение 0
x               // => 0: В выражениях имя переменной замещается ее значением.

// JavaScript поддерживает значения различных типов
x = 1;                    // Числа.
x = 0.01;                 // Целые и вещественные числа представлены одним типом.
x = "hello world";        // Строки текста в кавычках.
x = 'JavaScript';         // Строки можно также заключать в апострофы.
x = true;                 // Логические значения.
x = false;                // Другое логическое значение.
x = null;                 // null - особое значение, обозначающее "нет значения".
x = undefined;            // Значение undefined подобно значению null.
```

Двумя другими очень важными *типами* данных, которыми могут манипулировать программы на JavaScript, являются объекты и массивы. Они будут рассматриваться в главе 6 «Объекты» и в главе 7 «Массивы» однако они настолько важны, что вы не раз встретитесь с ними, прежде чем дойдете до этих глав.

```
// Наиболее важным типом данных в JavaScript являются объекты.
// Объект - это коллекция пар имя/значение или отображение строки в значение.
var book = {              // Объекты заключаются в фигурные скобки.
    topic: "JavaScript",  // Свойство "topic" имеет значение "JavaScript".
    fat: true             // Свойство "fat" имеет значение true.
};                        // Фигурная скобка отмечает конец объекта.

// Доступ к свойствам объектов выполняется с помощью . или []:
book.topic                // => "JavaScript"
book["fat"]               // => true: другой способ получить значение свойства.
book.author = "Flanagan"; // Создать новое свойство присваиванием.
book.contents = {};       // {} - пустой объект без свойств.

// JavaScript поддерживает массивы (списки с числовыми индексами) значений:
var primes = [2, 3, 5, 7]; // Массив из 4 значений, ограничивается [ и ].
primes[0]                 // => 2: первый элемент (с индексом 0) массива.
primes.length             // => 4: количество элементов в массиве.
primes[primes.length-1]   // => 7: последний элемент массива.
primes[4] = 9;            // Добавить новый элемент присваиванием.
```

```
primes[4] = 11;               // Или изменить значение имеющегося элемента.
var empty = [];               // [] - пустой массив без элементов.
empty.length                  // => 0

// Массивы и объекты могут хранить другие массивы и объекты:
var points = [                // Массив с 2 элементами.
    {x:0, y:0},               // Каждый элемент - это объект.
    {x:1, y:1}
];
var data = {                  // Объект с 2 свойствами
    trial1: [[1,2], [3,4]],   // Значение каждого свойства - это массив.
    trial2: [[2,3], [4,5]]    // Элементами массива являются массивы.
};
```

Синтаксические конструкции, представленные выше и содержащие списки элементов массивов в квадратных скобках или отображения свойств объектов в значения внутри фигурных скобок, часто называют *выражениями инициализации*, которые будут рассматриваться в главе 4 «Выражения и операторы». *Выражение* – это фраза на языке JavaScript, которую можно вычислить, чтобы получить значение. Например, применение `.` и `[]` для ссылки на значение свойства объекта или элемента массива является выражением. Возможно, вы заметили, что в листинге, приведенном выше, в строках, содержащих только выражение, комментарии начинаются со стрелки (=>), за которой следует значение выражения. Этому соглашению мы будем следовать на протяжении всей книги.

Наиболее типичным способом формирования выражений в JavaScript является использование *операторов*, подобно тому, как показано ниже:

```
// Операторы выполняют действия со значениями (операндами) и воспроизводят
// новое значение. Наиболее часто используемыми являются арифметические операторы:
3 + 2                       // => 5: сложение
3 - 2                       // => 1: вычитание
3 * 2                       // => 6: умножение
3 / 2                       // => 1.5: деление
points[1].x - points[0].x  // => 1: можно использовать более сложные операнды
"3" + "2"                   // => "32": + складывает числа, объединяет строки

// В JavaScript имеются некоторые сокращенные формы арифметических операторов
var count = 0;              // Объявление переменной
count++;                    // Увеличение значения переменной на 1
count--;                    // Уменьшение значения переменной на 1
count += 2;                 // Добавить 2: то же, что count = count + 2;
count *= 3;                 // Умножить на 3: то же, что count = count * 3;
count                       // => 6: имена переменных сами являются выражениями

// Операторы сравнения позволяют проверить два значения на равенство
// или неравенство, выяснить, какое значение меньше или больше, и т.д.
// Они возвращают значение true или false.
var x = 2, y = 3;           // Знаки = выполняют присваивание, а не сравнение
x == y                      // => false: равенство
x != y                      // => true: неравенство
x < y                       // => true: меньше
x <= y                      // => true: меньше или равно
x > y                       // => false: больше
x >= y                      // => false: больше или равно
"two" == "three"            // => false: две разных строки
```

```
"two" > "three"      // => true: при упорядочении по алфавиту строка "tw" больше, чем "th"
false == (x > y)     // => true: false равно false

// Логические операторы объединяют или инвертируют логические значения
(x == 2) && (y == 3)      // => true: оба сравнения истинны. && - "И"
(x > 3) || (y < 3)        // => false: оба сравнения ложны. || - "ИЛИ"
!(x == y)                 // => true: ! инвертирует логическое значение
```

Если фразы в языке JavaScript называются выражениями, то полные предложения называются *инструкциями*; они рассматриваются в главе 5 «Инструкции». В программном коде, приведенном выше, строки, заканчивающиеся точками с запятой, являются инструкциями. (В примере ниже можно увидеть инструкции, состоящие из нескольких строк, которые не завершаются точками с запятой.) Между инструкциями и выражениями много общего. Грубо говоря, выражение – это конструкция, которая вычисляет значение, но *ничего не делает*: она никак не изменяет состояние программы. Инструкции, напротив, не имеют значения (или их значение не представляет интереса для нас), но они изменяют состояние программы. Выше уже были показаны инструкции объявления переменных и присваивания значений. Еще одной обширной категорией инструкций являются *управляющие конструкции*, такие как условные инструкции и инструкции циклов. Примеры этих инструкций будут показаны далее, после того, как мы познакомимся с функциями.

Функция – это именованный и параметризованный блок программного кода JavaScript, который определяется один раз, а использоваться может многократно. Формальное знакомство с функциями мы отложим до главы 8 «Функции», однако, как и в случае с объектами и массивами, мы много раз встретимся с функциями, прежде чем доберемся до этой главы. Ниже приводятся несколько примеров простых функций:

```
// Функции - это параметризованные блоки программного кода JavaScript,
// которые можно вызывать многократно.
function plus1(x) {         // Определить функцию с именем "plus1" и с параметром "x"
    return x+1;             // Вернуть значение на 1 больше полученного
}                           // Функции заключаются в фигурные скобки

plus1(y)                    // => 4: y имеет значение 3, поэтому этот вызов вернет 3+1

var square = function(x) {  // Функции можно присваивать переменным
    return x*x;             // Вычислить значение функции
};                          // Точка с запятой отмечает конец присваивания.

square(plus1(y))            // => 16: вызов двух функций в одном выражении
```

При объединении функций с объектами получаются *методы*:

```
// Функции, присвоенные свойствам объектов, называются методами.
// Все объекты в JavaScript имеют методы:
var a = [];                 // Создать пустой массив
a.push(1,2,3);              // Метод push() добавляет элементы в массив
a.reverse();                // Другой метод: переставляет элементы в обратном порядке

// Можно определять собственные методы. Ключевое слово "this" ссылается на объект,
// в котором определен метод: в данном случае на массив points.
points.dist = function() {  // Метод вычисления расстояния между точками
    var p1 = this[0];       // Первый элемент массива, относительно которого вызван метод
    var p2 = this[1];       // Второй элемент объекта "this"
    var a = p2.x-p1.x;      // Разность координат X
    var b = p2.y-p1.y;      // Разность координат Y
    return Math.sqrt(a*a +  // Теорема Пифагора
                     b*b);  // Math.sqrt() вычисляет корень квадратный
};
points.dist()               // => 1.414: расстояние между 2-мя точками
```

Теперь, как было обещано, рассмотрим несколько функций, которые демонстрируют применение наиболее часто используемых управляющих инструкций JavaScript:

```
// В JavaScript имеются условные инструкции и инструкции циклов, синтаксически
// похожие на аналогичные инструкции C, C++, Java и в других языках.
function abs(x) {           // Функция, вычисляющая абсолютное значение
    if (x >= 0) {           // Инструкция if ...
        return x;           // выполняет этот код, если сравнение дает true.
    }                       // Конец предложения if.
    else {                  // Необязательное предложение else выполняет свой код,
        return -x;          // если сравнение дает значение false.
    }                       // Фигурные скобки можно опустить, если предложение
                            // содержит 1 инструкцию.
}                           // Обратите внимание на инструкции return внутри if/else.

function factorial(n) {     // Функция, вычисляющая факториал
    var product = 1;        // Начать с произведения, равного 1
    while(n > 1) {          // Повторять инструкции в {}, пока выраж. в () истинно
        product *= n;       // Сокращенная форма выражения product = product * n;
        n--;                // Сокращенная форма выражения n = n - 1
    }                       // Конец цикла
    return product;         // Вернуть произведение
}
factorial(4)                // => 24: 1*4*3*2

function factorial2(n) {    // Другая версия, использующая другой цикл
    var i, product = 1;     // Начать с 1
    for(i=2; i <= n; i++)   // i автоматически увеличивается с 2 до n
        product *= i;       // Выполнять в каждом цикле. {} можно опустить,
                            // если тело цикла состоит из 1 инструкции
    return product;         // Вернуть факториал
}
factorial2(5)               // => 120: 1*2*3*4*5
```

JavaScript – объектно-ориентированный язык, но используемая в нем объектная модель в корне отличается от модели, используемой в большинстве других языков. В главе 9 «Классы и модули» детально рассматривается объектно-ориентированное программирование на языке JavaScript на большом количестве примеров; эта глава является одной из самых больших в книге. Ниже приводится очень простой пример, демонстрирующий определение класса JavaScript для представления точек на плоскости. Объекты, являющиеся экземплярами этого класса, обладают единственным методом с методом `r()`, который вычисляет расстояние между данной точкой и началом координат:

```
// Определение функции-конструктора для инициализации нового объекта Point
function Point(x,y) { // По соглашению имя конструкторов начинается с заглавного символа
    this.x = x;       // this - ссылка на инициализируемый объект
    this.y = y;             // Сохранить аргументы в свойствах объекта
}                           // Ничего возвращать не требуется

// Чтобы создать новый экземпляр, необходимо вызвать функцию-конструктор
// с ключевым словом "new"
var p = new Point(1, 1);  // Точка на плоскости с координатами (1,1)

// Методы объектов Point определяются за счет присваивания функций свойствам
// объекта-прототипа, ассоциированного с функцией-конструктором.
Point.prototype.r = function() {
    return Math.sqrt(     // Вернуть корень квадратный от x² + y²
        this.x * this.x + // this - это объект Point, относительно которого...
        this.y * this.y   // ...вызывается метод.
    );
};

// Теперь объект p типа Point (и все последующие объекты Point) наследует метод r()
p.r()                     // => 1.414...
```

Глава 9 является кульминацией первой части, а главы, которые следуют за ней, связывают некоторые оборванные концы и завершают исследование базового языка. В главе 10 «Шаблоны и регулярные выражения» описывается грамматика регулярных выражений и демонстрируются приемы использования регулярных выражений для реализации сопоставления с текстовыми шаблонами. В главе 11 «Подмножества и расширения JavaScript» рассматриваются подмножества и расширения базового языка JavaScript. Наконец, прежде чем перейти к исследованию клиентского JavaScript в веб-броузерах, в главе 12 «Серверный JavaScript» будут представлены два способа использования JavaScript за пределами веб-броузеров.

1.2. Клиентский JavaScript

Изучение клиентского JavaScript представляет собой задачу, нелинейную из-за перекрестных ссылок в значительно меньшей мере, чем базовый язык, и поэтому вполне возможно изучать особенности использования JavaScript в веб-броузерах в линейном порядке. Возможно, вы взялись за чтение этой книги, чтобы изучить клиентский JavaScript – тему далекой второй части, поэтому здесь мы приводим краткий обзор основных приемов программирования клиентских сценариев, который сопровождается подробным примером.

Глава 13 «JavaScript в веб-броузерах» является первой главой второй части, в которой описываются детали использования JavaScript в веб-броузерах. Самое важное, что вы узнаете в этой главе, – программный код JavaScript может встраиваться в HTML-файлы с помощью тега `<script>`:

```
<html>
<head>
<script src="library.js"></script> <!-- подключить библиотеку JavaScript -->
</head>
<body>
<p>Это абзац HTML</p>
<script>
// Это некоторый программный код на клиентском JavaScript,
// встроенный непосредственно в HTML-файл
</script>
```

```
<p>Далее опять следует разметка HTML.</p>
</body>
</html>
```

Глава 14 «Объект Window» исследует приемы управления веб-броузером и описывает некоторые наиболее важные глобальные функции клиентского JavaScript. Например:

```
<script>
function moveon() {
    // Вывести модальный диалог, чтобы получить ответ пользователя
    var answer = confirm("Ready to move on?");
    // Если пользователь щелкнул на кнопке "OK", заставить броузер загрузить новую страницу
    if (answer) window.location = "http://google.com";
}
// Запустить функцию, объявленную выше, через 1 минуту (60000 миллисекунд).
setTimeout(moveon, 60000);
</script>
```

Обратите внимание, что примеры программного кода на клиентском JavaScript в этом разделе длиннее примеров на базовом языке, которые мы видели выше в этой главе. Эти примеры не предназначены для ввода в окне консоли Firebug (или в другом подобном инструменте). Однако вы можете вставлять их в HTML-файлы и затем запускать, открывая файлы в веб-броузере. Так, пример, приведенный выше, является самостоятельным HTML-файлом.

Глава 15 «Работа с документами» переходит к исследованию фактической работы, выполняемой с помощью JavaScript на стороне клиента, – управлению содержимым документа HTML. Она покажет вам, как выбирать определенные элементы HTML из документов, как устанавливать HTML-атрибуты этих элементов, как изменять содержимое элементов и как добавлять в документ новые элементы. Следующая функция демонстрирует некоторые из простейших приемов поиска и изменения элементов документа:

```
// Выводит сообщение в специальной области для отладочных сообщений внутри документа.
// Если документ не содержит такой области, она создается.
function debug(msg) {
    // Отыскать область для отладочных сообщений в документе, поиск по HTML-атрибуту id
    var log = document.getElementById("debuglog");

    // Если элемент с атрибутом id="debuglog" отсутствует, создать его.
    if (!log) {
        log = document.createElement("div");  // Создать элемент <div>
        log.id = "debuglog";                  // Установить атрибут id
        log.innerHTML = "<h1>Debug Log</h1>"; // Начальное содержимое
        document.body.appendChild(log);       // Добавить в конец документа
    }

    // Теперь обернуть сообщение в теги <pre> и добавить в элемент log
    var pre = document.createElement("pre");  // Создать тег <pre>
    var text = document.createTextNode(msg);  // Обернуть msg в текстовый узел
    pre.appendChild(text);                    // Добавить текст в тег <pre>
    log.appendChild(pre);                     // Добавить <pre> в элемент log
}
```

Глава 15 демонстрирует, как с помощью JavaScript можно управлять HTML-элементами, которые определяют содержимое веб-страниц. Глава 16 «Каскадные таб-

лицы стилей» демонстрирует, как с помощью JavaScript можно управлять каскадными таблицами стилей CSS, определяющими представление содержимого. Чаще всего для этой цели используется атрибут HTML-элементов `style` или `class`:

```
function hide(e, reflow) {          // Скрывает элемент e, изменяя его стиль
    if (reflow) {                   // Если 2-й аргумент true,
        e.style.display = "none"    // скрыть элемент и использовать
    }                               // занимаемое им место
    else {                          // Иначе
        e.style.visibility = "hidden"; // сделать e невидимым, но оставить
    }                               // занимаемое им место пустым
}

function highlight(e) { // Выделяет e, устанавливая класс CSS
    // Просто добавляет или переопределяет HTML-атрибут class.
    // Предполагается, что таблица стилей CSS уже содержит определение класса "hilite"
    if (!e.className) e.className = "hilite";
    else e.className += " hilite";
}
```

JavaScript позволяет не только управлять содержимым и оформлением HTML-документов в броузерах, но и определять поведение этих документов с помощью *обработчиков событий*. Обработчик событий – это функция JavaScript, которая регистрируется в броузере и вызывается броузером, когда возникает событие определенного типа. Таким событием может быть щелчок мышью или нажатие клавиши (или какое-то движение двумя пальцами на экране смартфона). Обработчик события может также вызываться броузером по окончании загрузки документа, при изменении размеров окна броузера или при вводе данных в элемент HTML-формы. Глава 17 «Обработка событий» описывает, как определять и регистрировать обработчики событий и как вызываются эти обработчики при появлении событий.

Простейший способ объявления обработчиков событий заключается в использовании HTML-атрибутов, имена которых начинаются с приставки «on». Обработчик «onclick» особенно удобен при создании простых тестовых программ. Предположим, что вы сохранили функции `debug()` и `hide()`, представленные выше, в файлах с именами *debug.js* и *hide.js*. В этом случае можно было бы написать простой тестовый HTML-файл, использующий элементы `<button>` с атрибутами `onclick`, определяющими обработчики событий:

```
<script src="debug.js"></script>
<script src="hide.js"></script>
Hello
<button onclick="hide(this,true); debug('hide button 1');">Hide1</button>
<button onclick="hide(this); debug('hide button 2');">Hide2</button>
World
```

Ниже приводится еще один пример программного кода на клиентском JavaScript, использующего механизм событий. Он регистрирует обработчик очень важного события «load» и дополнительно демонстрирует более сложный способ регистрации обработчика события «click»:

```
// Событие "load" возбуждается, когда документ будет полностью загружен.
// Обычно мы вынуждены ждать этого события, прежде чем можно будет запустить
// наш программный код JavaScript.
window.onload = function() { // Запустит функцию после загрузки документа
```

```
    // Отыскать все теги <img> в документе
    var images = document.getElementsByTagName("img");

    // Обойти их все в цикле, добавить к каждому обработчик события "click",
    // чтобы обеспечить сокрытие любого изображения после щелчка на нем.
    for(var i = 0; i < images.length; i++) {
        var image = images[i];
        if (image.addEventListener)   // Другой способ регистрации обработчика
            image.addEventListener("click", hide, false);
        else                          // Для совместимости с версией IE8 и ниже
            image.attachEvent("onclick", hide);
    }

    // Это функция-обработчик событий, которая регистрируется выше
    function hide(event) { event.target.style.visibility = "hidden"; }
};
```

Главы 15, 16 и 17 описывают, как с помощью JavaScript управлять содержимым (HTML), представлением (CSS) и поведением (обработка событий) веб-страниц. Прикладной интерфейс, описываемый в этих главах, является достаточно сложным, и до недавнего времени испытывал проблемы с совместимостью между броузерами. По этим причинам многие или большинство программистов на клиентском JavaScript предпочитают использовать клиентские библиотеки или фреймворки, упрощающие программирование. Наиболее популярна из этих библиотек – библиотека jQuery, которая обсуждается в главе 19 «Библиотека jQuery». Библиотека jQuery определяет простой и удобный программный интерфейс для управления содержимым документа, его представлением и поведением. Она была тщательно протестирована и может использоваться во всех основных броузерах, включая довольно старые, такие как IE6.

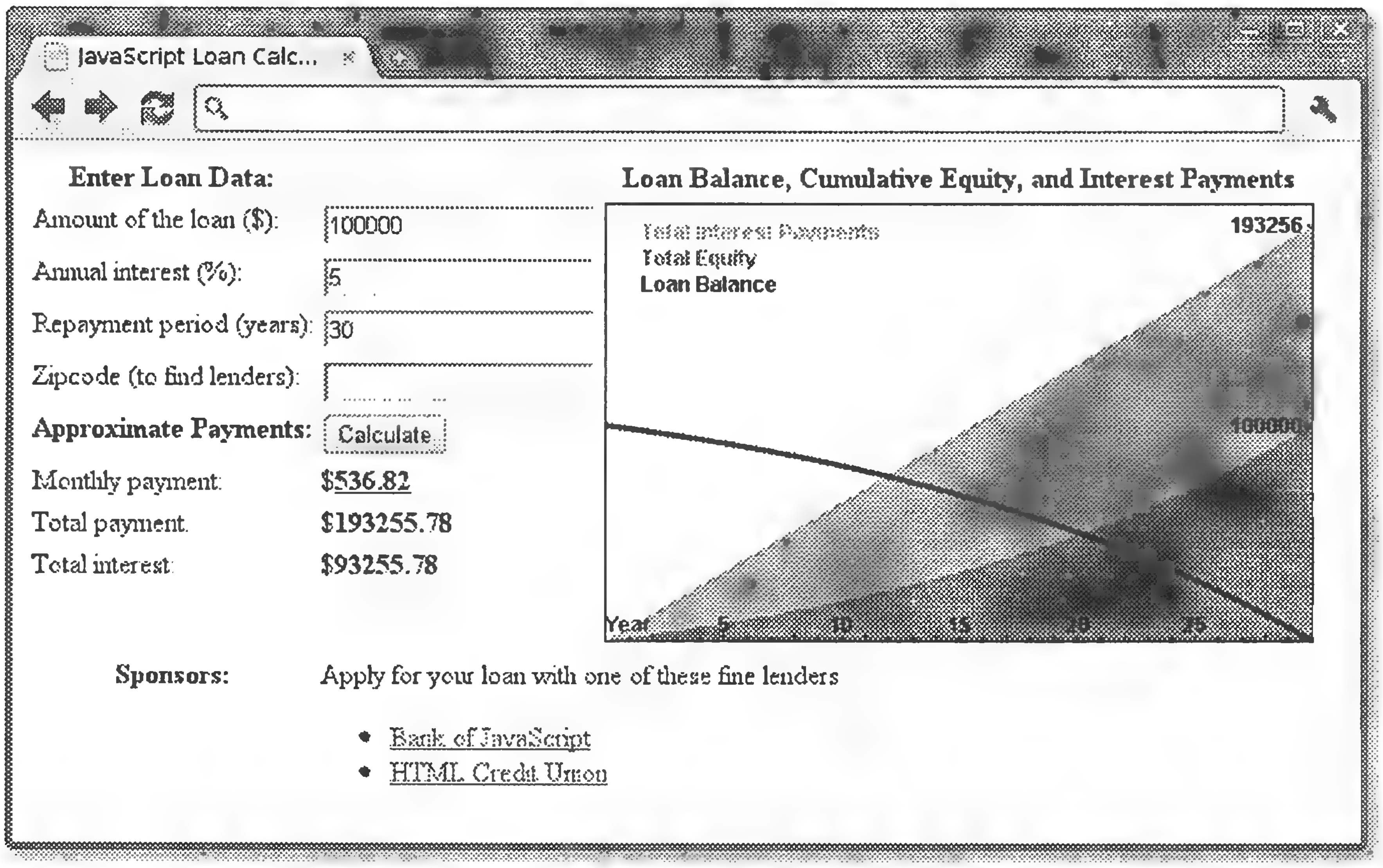
Программный код, использующий jQuery, легко отличить по частому использованию функции `$()`. Ниже показано, как будет выглядеть функция `debug()`, представленная выше, если переписать ее с использованием jQuery:

```
function debug(msg) {
    var log = $("#debuglog");          // Отыскать элемент для вывода msg.
    if (log.length == 0) {             // Если отсутствует, создать его...
        log = $("<div id='debuglog'><h1>Debug Log</h1></div>");
        log.appendTo(document.body);   // и вставить в конец тела документа.
    }
    log.append($("<pre/>").text(msg)); // Завернуть msg в тег <pre>
}                                      // и добавить в элемент log
```

В этих четырех главах из второй части в действительности рассматривается все, что касается *веб-страниц*. Другие четыре главы переключают внимание на *веб-приложения*. Эти главы не о том, как использовать веб-броузеры для отображения документов, содержимое, представление и поведение которых управляется с помощью JavaScript. Они рассказывают об использовании веб-броузеров как прикладной платформы и описывают прикладной интерфейс, предоставляемый современными броузерами для поддержки сложных, современных клиентских веб-приложений. Глава 18 «Работа с протоколом HTTP» описывает, как с помощью JavaScript можно управлять HTTP-запросами – своего рода сетевой прикладной интерфейс. Глава 20 «Сохранение данных на стороне клиента» описывает механизмы, позволяющие сохранять данные (и даже целые приложения) на стороне клиента для использования в последующих сеансах работы. Глава 21 «Работа с графикой и медиафайлами на стороне клиента» охватывает клиентский прикладной интерфейс, позволяющий создавать произвольные графические изображения в HTML-теге `<canvas>`. И наконец, глава 22 «Прикладные интерфейсы HTML5» охватывает новые прикладные интерфейсы веб-приложений, определяемые или принятые стандартом HTML5. Сетевые взаимодействия, организация хранения данных, работа с графикой – все эти службы операционных систем, доступные посредством веб-броузеров, образуют новую, платформонезависимую среду выполнения приложений. Если вы нацелены на броузеры, которые поддерживают эти новые прикладные интерфейсы, то сейчас наступает самое интересное время для программистов на клиентском JavaScript. Здесь не приводятся примеры программного кода из этих заключительных четырех глав, однако расширенный пример, представленный ниже, использует некоторые из этих новых прикладных интерфейсов.

1.2.1. Пример: калькулятор платежей по ссуде на JavaScript

Эта глава завершается расширенным примером, объединяющим в себе многие из описанных выше приемов и демонстрирующим полноценную программу на клиентском JavaScript (плюс HTML и CSS). В примере 1.1 представлена реализация простого калькулятора для вычисления платежей по ссуде (рис. 1.2).



Enter Loan Data:

Amount of the loan ($):	100000
Annual interest (%):	5
Repayment period (years):	30
Zipcode (to find lenders):	
Approximate Payments:	
Monthly payment:	$536.82
Total payment:	$193255.78
Total interest:	$93255.78

Sponsors: Apply for your loan with one of these fine lenders

- Bank of JavaScript
- HTML Credit Union

Рис. 1.2. Веб-приложение – калькулятор платежей по ссуде

Стоит потратить время на внимательное рассмотрение примера 1.1. Вряд ли вы сумеете досконально разобраться в нем, однако благодаря подробным комментариям вы должны по крайней мере получить общее представление о том, как действует это веб-приложение. Пример демонстрирует множество особенностей базового языка JavaScript, а также некоторые важные приемы программирования на клиентском JavaScript:

- **Поиск элементов в документе.**
- **Получение ввода пользователя с помощью элементов форм.**
- **Изменение содержимого элементов документа.**
- **Сохранение данных в броузере.**
- **Управление HTTP-запросами.**
- **Создание графики с помощью элемента** `<canvas>`**.**

Пример 1.1. Калькулятор вычисления платежей по ссуде на JavaScript

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Loan Calculator</title>
<style> /* Таблица стилей CSS: определяет внешний вид вывода программы */
.output { font-weight: bold; }          /* Жирный шрифт для вычисленных значений */
#payment { text-decoration: underline; } /* Для элементов с id="payment" */
#graph { border: solid black 1px; }     /* Простая рамка для диаграммы */
th, td { vertical-align: top; }         /* Выравнивание в ячейках таблицы */
</style>
</head>
<body>
<!--
  Это HTML-таблица с элементами <input>, позволяющими вводить данные, и с элементами
  <span>, в которых отображаются результаты вычислений. Эти элементы имеют идентификаторы,
  такие как "interest" и "years". Данные идентификаторы используются в JavaScript-коде,
  который следует за определением таблицы. Обратите внимание, что для некоторых элементов
  ввода определены обработчики событий "onchange" и "onclick". В них заданы
  строки JavaScript-кода, выполняемого при вводе данных или щелчке на кнопке.
-->
<table>
  <tr><th>Enter Loan Data:</th>
    <td></td>
    <th>Loan Balance, Cumulative Equity, and Interest Payments</th></tr>
  <tr><td>Amount of the loan ($):</td>
    <td><input id="amount" onchange="calculate();"></td>
    <td rowspan=8>
      <canvas id="graph" width="400" height="250"></canvas></td></tr>
  <tr><td>Annual interest (%):</td>
    <td><input id="apr" onchange="calculate();"></td></tr>
  <tr><td>Repayment period (years):</td>
    <td><input id="years" onchange="calculate();"></td>
  <tr><td>Zipcode (to find lenders):</td>
    <td><input id="zipcode" onchange="calculate();"></td>
  <tr><th>Approximate Payments:</th>
    <td><button onclick="calculate();">Calculate</button></td></tr>
  <tr><td>Monthly payment:</td>
    <td>$<span class="output" id="payment"></span></td></tr>
  <tr><td>Total payment:</td>
    <td>$<span class="output" id="total"></span></td></tr>
  <tr><td>Total interest:</td>
    <td>$<span class="output" id="totalinterest"></span></td></tr>
  <tr><th>Sponsors:</th><td colspan=2>
    Apply for your loan with one of these fine lenders:
```

```
    <div id="lenders"></div></td></tr>
</table>

<!-- Остальная часть примера - JavaScript-код в теге <script> ниже. Обычно сценарии -->
<!-- помещаются в начало документа, в заголовок <head>, но в данном случае вам проще -->
<!-- будет понять пример, если JavaScript-код будет находиться ниже HTML-содержимого. -->
<script>
"use strict"; // Использовать строгий режим ECMAScript 5, если броузер поддерживает его
/*
 * Этот сценарий определяет функцию calculate(), вызываемую обработчиками событий
 * в разметке HTML выше. Функция читает значения из элементов <input>, вычисляет размеры
 * платежей по ссуде, отображает результаты в элементах <span>. Кроме того, она сохраняет
 * пользовательские данные, отображает ссылки на кредитные учреждения и рисует диаграмму.
 */
function calculate() {
    // Отыскать элементы ввода и вывода в документе
    var amount = document.getElementById("amount");
    var apr = document.getElementById("apr");
    var years = document.getElementById("years");
    var zipcode = document.getElementById("zipcode");
    var payment = document.getElementById("payment");
    var total = document.getElementById("total");
    var totalinterest = document.getElementById("totalinterest");

    // Получить ввод пользователя из элементов ввода. Предполагается, что все данные
    // являются корректными. Преобразовать процентную ставку из процентов
    // в десятичное число и преобразовать годовую ставку в месячную ставку.
    // Преобразовать период платежей в годах в количество месячных платежей.
    var principal = parseFloat(amount.value);
    var interest = parseFloat(apr.value) / 100 / 12;
    var payments = parseFloat(years.value) * 12;

    // Теперь вычислить сумму ежемесячного платежа.
    var x = Math.pow(1 + interest, payments); // Math.pow() вычисляет степень
    var monthly = (principal*x*interest)/(x-1);

    // Если результатом является конечное число, следовательно, пользователь
    // указал корректные данные и результаты можно отобразить
    if (isFinite(monthly)) {
        // Заполнить поля вывода, округлив результаты до 2 десятичных знаков
        payment.innerHTML = monthly.toFixed(2);
        total.innerHTML = (monthly * payments).toFixed(2);
        totalinterest.innerHTML = ((monthly*payments)-principal).toFixed(2);

        // Сохранить ввод пользователя, чтобы можно было восстановить данные
        // при следующем открытии страницы
        save(amount.value, apr.value, years.value, zipcode.value);

        // Реклама: отыскать и отобразить ссылки на сайты местных
        // кредитных учреждений, но игнорировать сетевые ошибки
        try { // Перехватывать все ошибки, возникающие в этих фигурных скобках
            getLenders(amount.value, apr.value, years.value, zipcode.value);
        }
        catch(e) { /* И игнорировать эти ошибки */ }

        // В заключение вывести график изменения остатка по кредиту, а также
        // графики сумм, выплачиваемых в погашение кредита и по процентам
        chart(principal, interest, monthly, payments);
    }
    else {
        // Результат не является числом или имеет бесконечное значение,
        // что означает, что были получены неполные или некорректные данные.
        // Очистить все результаты, выведенные ранее.
        payment.innerHTML = ""; // Стереть содержимое этих элементов
        total.innerHTML = ""
        totalinterest.innerHTML = "";
        chart();                // При вызове без аргументов очищает диаграмму
    }
}

// Сохранить ввод пользователя в свойствах объекта localStorage. Значения этих свойств
// будут доступны при повторном посещении страницы. В некоторых броузерах (например,
// в Firefox) возможность сохранения не поддерживается, если страница открывается
// с адресом URL вида file://. Однако она поддерживается при открытии страницы через HTTP.
function save(amount, apr, years, zipcode) {
    if (window.localStorage) { // Выполнить сохранение, если поддерживается
        localStorage.loan_amount = amount;
        localStorage.loan_apr = apr;
        localStorage.loan_years = years;
        localStorage.loan_zipcode = zipcode;
    }
}

// Автоматически восстановить поля ввода при загрузке документа.
window.onload = function() {
    // Если броузер поддерживает localStorage и имеются сохраненные данные
    if (window.localStorage && localStorage.loan_amount) {
        document.getElementById("amount").value = localStorage.loan_amount;
        document.getElementById("apr").value = localStorage.loan_apr;
        document.getElementById("years").value = localStorage.loan_years;
        document.getElementById("zipcode").value = localStorage.loan_zipcode;
    }
};

// Передать ввод пользователя серверному сценарию, который может (теоретически) возвращать
// список ссылок на сайты местных кредитных учреждений, готовых предоставить кредит.
// Данный пример не включает фактическую реализацию такого сценария поиска кредитных
// учреждений. Но если такой сценарий уже имеется, данная функция могла бы работать с ним.
function getLenders(amount, apr, years, zipcode) {
    // Если броузер не поддерживает объект XMLHttpRequest, не делать ничего
    if (!window.XMLHttpRequest) return;

    // Отыскать элемент для отображения списка кредитных учреждений
    var ad = document.getElementById("lenders");
    if (!ad) return;                           // Выйти, если элемент отсутствует

    // Преобразовать ввод пользователя в параметры запроса в строке URL
    var url = "getLenders.php" +               // Адрес URL службы плюс
        "?amt=" + encodeURIComponent(amount) + // данные пользователя
        "&apr=" + encodeURIComponent(apr) +    // в строке запроса
        "&yrs=" + encodeURIComponent(years) +
        "&zip=" + encodeURIComponent(zipcode);

    // Получить содержимое по заданному адресу URL с помощью XMLHttpRequest
    var req = new XMLHttpRequest();            // Создать новый запрос
    req.open("GET", url);                      // Указать тип запроса HTTP GET для url
    req.send(null);                                 // Отправить запрос без тела

    // Перед возвратом зарегистрировать обработчик события, который будет вызываться
    // при получении HTTP-ответа от сервера. Такой прием асинхронного программирования
    // является довольно обычным в клиентском JavaScript.
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.status == 200) {
            // Если мы попали сюда, следовательно, был получен корректный HTTP-ответ
            var response = req.responseText;       // HTTP-ответ в виде строки
            var lenders = JSON.parse(response);     // Преобразовать в JS-массив

            // Преобразовать массив объектов lender в HTML-строку
            var list = "";
            for(var i = 0; i < lenders.length; i++) {
                list += "<li><a href='" + lenders[i].url + "'>" +
                    lenders[i].name + "</a>";
            }

            // Отобразить полученную HTML-строку в элементе,
            // ссылка на который была получена выше.
            ad.innerHTML = "<ul>" + list + "</ul>";
        }
    }
}

// График помесячного изменения остатка по кредиту, а также графики сумм,
// выплачиваемых в погашение кредита и по процентам в HTML-элементе <canvas>.
// Если вызывается без аргументов, просто очищает ранее нарисованные графики.
function chart(principal, interest, monthly, payments) {
    var graph = document.getElementById("graph");   // Ссылка на тег <canvas>
    graph.width = graph.width;                      // Магия очистки элемента canvas

    // Если функция вызвана без аргументов или броузер не поддерживает
    // элемент <canvas>, то просто вернуть управление.
    if (arguments.length == 0 || !graph.getContext) return;

    // Получить объект "контекста" для элемента <canvas>,
    // который определяет набор методов рисования
    var g = graph.getContext("2d"); // Рисование выполняется с помощью этого объекта
    var width = graph.width, height = graph.height; // Получить размер холста

    // Следующие функции преобразуют количество месячных платежей
    // и денежные суммы в пикселы
    function paymentToX(n) { return n * width/payments; }
    function amountToY(a) { return height-(a*height/(monthly*payments*1.05));}

    // Платежи - прямая линия из точки (0,0) в точку (payments,monthly*payments)
    g.moveTo(paymentToX(0), amountToY(0));          // Из нижнего левого угла
    g.lineTo(paymentToX(payments),                  // В правый верхний
             amountToY(monthly*payments));
    g.lineTo(paymentToX(payments), amountToY(0));   // В правый нижний
    g.closePath();                                  // И обратно в начало
    g.fillStyle = "#f88";                           // Светло-красный
    g.fill();                                       // Залить треугольник
    g.font = "bold 12px sans-serif";                // Определить шрифт
    g.fillText("Total Interest Payments", 20,20);   // Вывести текст в легенде

    // Кривая накопленной суммы погашения кредита не является линейной
    // и вывод ее реализуется немного сложнее
    var equity = 0;
    g.beginPath();                                 // Новая фигура
    g.moveTo(paymentToX(0), amountToY(0));         // из левого нижнего угла
    for(var p = 1; p <= payments; p++) {
        // Для каждого платежа выяснить долю выплат по процентам
        var thisMonthsInterest = (principal-equity)*interest;
        equity += (monthly - thisMonthsInterest);  // Остаток - погашение кред.
        g.lineTo(paymentToX(p),amountToY(equity)); // Линию до этой точки
    }
    g.lineTo(paymentToX(payments), amountToY(0));  // Линию до оси X
    g.closePath();                                 // И опять в нач. точку
    g.fillStyle = "green";                         // Зеленый цвет
    g.fill();                                      // Залить обл. под кривой
    g.fillText("Total Equity", 20,35);             // Надпись зеленым цветом

    // Повторить цикл, как выше, но нарисовать график остатка по кредиту
    var bal = principal;
    g.beginPath();
    g.moveTo(paymentToX(0),amountToY(bal));
    for(var p = 1; p <= payments; p++) {
        var thisMonthsInterest = bal*interest;
        bal -= (monthly - thisMonthsInterest);     // Остаток от погаш. по кредиту
        g.lineTo(paymentToX(p),amountToY(bal));    // Линию до этой точки
    }
    g.lineWidth = 3;                               // Жирная линия
    g.stroke();                                    // Нарисовать кривую графика
    g.fillStyle = "black";                         // Черный цвет для текста
    g.fillText("Loan Balance", 20,50);             // Элемент легенды

    // Нарисовать отметки лет на оси X
    g.textAlign="center";                          // Текст меток по центру
    var y = amountToY(0);                          // Координата Y на оси X
    for(var year=1; year*12 <= payments; year++) { // Для каждого года
        var x = paymentToX(year*12);               // Вычислить позицию метки
        g.fillRect(x-0.5,y-3,1,3);                 // Нарисовать метку
        if (year == 1) g.fillText("Year", x, y-5); // Подписать ось
        if (year % 5 == 0 && year*12 !== payments) // Числа через каждые 5 лет
            g.fillText(String(year), x, y-5);
    }

    // Суммы платежей у правой границы
    g.textAlign = "right";                         // Текст по правому краю
    g.textBaseline = "middle";                     // Центрировать по вертикали
    var ticks = [monthly*payments, principal];     // Вывести две суммы
    var rightEdge = paymentToX(payments);          // Координата X на оси Y
    for(var i = 0; i < ticks.length; i++) {        // Для каждой из 2 сумм
        var y = amountToY(ticks[i]);               // Определить координату Y
        g.fillRect(rightEdge-3, y-0.5, 3,1);       // Нарисовать метку
        g.fillText(String(ticks[i].toFixed(0)),    // И вывести рядом сумму.
                   rightEdge-5, y);
    }
}
</script>
</body>
</html>
```

I

Базовый JavaScript

Данная часть книги включает главы со 2 по 12, она описывает базовый язык JavaScript и задумана как справочник по языку JavaScript. Прочитав главы этой части один раз, вы, возможно, будете неоднократно возвращаться к ним, чтобы освежить в памяти более сложные особенности языка.

- Глава 2 «Лексическая структура»
- Глава 3 «Типы данных, значения и переменные»
- Глава 4 «Выражения и операторы»
- Глава 5 «Инструкции»
- Глава 6 «Объекты»
- Глава 7 «Массивы»
- Глава 8 «Функции»
- Глава 9 «Классы и модули»
- Глава 10 «Шаблоны и регулярные выражения»
- Глава 11 «Подмножества и расширения JavaScript»
- Глава 12 «Серверный JavaScript»

2

Лексическая структура

Лексическая структура языка программирования – это набор элементарных правил, определяющих, как пишутся программы на этом языке. Это низкоуровневый синтаксис языка; он определяет вид имен переменных, символы, используемые для обозначения комментариев, и то, как одна инструкция отделяется от другой. Эта короткая глава описывает лексическую структуру JavaScript.

2.1. Набор символов

При написании программ на JavaScript используется набор символов Юникода. Юникод является надмножеством кодировок ASCII и Latin-1 и поддерживает практически все письменные языки, имеющиеся на планете. Стандарт ECMAScript 3 требует, чтобы реализации JavaScript обеспечивали поддержку стандарта Юникода версии 2.1 или выше, а стандарт ECMAScript 5 требует, чтобы реализации обеспечивали поддержку стандарта Юникода версии 3 или выше. Более подробно о Юникоде и JavaScript говорится во врезке в разделе 3.2.

2.1.1. Чувствительность к регистру

JavaScript – это язык, чувствительный к регистру символов. Это значит, что ключевые слова, имена переменных и функций и любые другие *идентификаторы* языка должны всегда содержать одинаковые наборы прописных и строчных букв. Например, ключевое слово `while` должно набираться как «while», а не «While» или «WHILE». Аналогично `online`, `Online`, `OnLine` и `ONLINE` – это имена четырех разных переменных.

Заметим, однако, что язык разметки HTML (в отличие от XHTML) не чувствителен к регистру. Так как HTML и клиентский JavaScript тесно связаны, это различие может привести к путанице. Многие JavaScript-объекты и их свойства имеют те же имена, что и теги и атрибуты языка HTML, которые они обозначают. Однако если в HTML эти теги и атрибуты могут набираться в любом регистре, то в JavaScript они обычно должны набираться строчными буквами. Например, атрибут `onclick` обработчика события чаще всего задается в HTML как `onClick`, однако в JavaScript-коде (или в XHTML-документе) он должен быть обозначен как `onclick`.

2.1.2. Пробелы, переводы строк и символы управления форматом

JavaScript игнорирует пробелы, которые могут присутствовать между лексемами в программе. Кроме того, JavaScript также по большей части игнорирует символы перевода строки (за одним исключением, о котором рассказывается в разделе 2.5). Поэтому пробелы и символы перевода строки могут без ограничений использоваться в исходных текстах программ для форматирования и придания им удобочитаемого внешнего вида.

Помимо обычного символа пробела (\u0020) JavaScript дополнительно распознает как пробельные следующие символы: табуляция (\u0009), вертикальная табуляция (\u000B), перевод формата (\u000C), неразрывный пробел (\u00A0), маркер порядка следования байтов (\uFEFF), а также все символы Юникода, относящиеся к категории Zs. Следующие символы распознаются интерпретаторами JavaScript как символы конца строки: перевод строки (\u000A), возврат каретки (\u000D), разделитель строк (\u2028) и разделитель абзацев (\u2029). Последовательность из символов возврата каретки и перевода строки интерпретируется как единственный символ завершения строки.

Символы Юникода, управляющие форматом (категория Cf), такие как RIGHT-TO-LEFT MARK (\u200F) и LEFT-TO-RIGHT MARK (\u200E), управляют визуальным представлением текста, в котором они присутствуют. Они имеют большое значение для корректного отображения текста на некоторых языках и являются допустимыми в комментариях JavaScript, строковых литералах и в литералах регулярных выражений, но не в идентификаторах (таких как имена переменных), определяемых в программах JavaScript. Исключение составляют ZERO WIDTH JOINER (\u200D) и ZERO WIDTH NON-JOINER (\u200C), которые можно использовать в идентификаторах при условии, что они не являются первыми символами идентификаторов. Как отмечалось выше, символ управления порядком следования байтов (\uFEFF) интерпретируется как пробельный символ.

2.1.3. Экранированные последовательности Юникода

Некоторые компьютеры и программное обеспечение не могут отображать или обеспечивать ввод полного набора символов Юникода. Для поддержки программистов, использующих подобную устаревшую технику, JavaScript определяет специальные последовательности, состоящие из шести символов ASCII, представляющие 16-битные кодовые пункты Юникода. Эти экранированные последовательности Юникода начинаются с символов \u, за которыми следуют точно четыре шестнадцатеричные цифры (при этом символы A–F могут быть и строчными, и прописными). Экранированные последовательности Юникода могут появляться в строковых литералах JavaScript, в литералах регулярных выражений и в идентификаторах (но не в ключевых словах языка). Экранированная последовательность Юникода для символа é, например, имеет вид \u00E9, и с точки зрения JavaScript следующие две строки являются идентичными:

```
"café" === "caf\u00e9" // => true
```

Экранированные последовательности Юникода могут также появляться в комментариях, но поскольку комментарии игнорируются, в данном контексте они воспринимаются как последовательность символов ASCII и не интерпретируются как символы Юникода.

2.1.4. Нормализация

Юникод позволяет закодировать один и тот же символ несколькими способами. Строка «é», например, может быть закодирована как единственный символ Юникода `\u00E9` или как обычный ASCII-символ е, со следующим за ним диакритическим знаком `\u0301`. Эти два способа представления обеспечивают одинаковое отображение в текстовом редакторе, но имеют различные двоичные коды и с точки зрения компьютера считаются различными. Стандарт Юникода определяет предпочтительные способы кодирования для всех символов и задает процедуру нормализации для приведения текста к канонической форме, пригодной для сравнения. Интерпретаторы JavaScript полагают, что интерпретируемый программный код уже был нормализован, и не предпринимают никаких попыток нормализовать идентификаторы, строки или регулярные выражения.

2.2. Комментарии

JavaScript поддерживает два способа оформления комментариев. Любой текст между символами `//` и концом строки рассматривается как комментарий и игнорируется JavaScript. Любой текст между символами `/*` и `*/` также рассматривается как комментарий. Эти комментарии могут состоять из нескольких строк, но не могут быть вложенными. Следующие строки представляют собой корректные JavaScript-комментарии:

```
// Это однострочный комментарий.
/* Это тоже комментарий */ // а это другой комментарий.
/*
 * Это еще один комментарий.
 * Он располагается в нескольких строках.
 */
```

2.3. Литералы

Литерал – это значение, указанное непосредственно в тексте программы. Ниже приводятся примеры различных литералов:

```
12             // Число двенадцать
1.2            // Число одна целая две десятых
"hello world"  // Строка текста
'Hi'           // Другая строка
true           // Логическое значение
false          // Другое логическое значение
/javascript/gi // Литерал "регулярного выражения" (для поиска по шаблону)
null           // Пустой объект
```

Полное описание числовых и строковых литералов приводится в главе 3. Литералы регулярных выражений рассматриваются в главе 10. Ниже приводятся более сложные выражения (смотрите раздел 4.2), которые могут служить литералами массивов и объектов:

```
{ x:1, y:2 } // Инициализатор объекта
[1,2,3,4,5] // Инициализатор массива
```

2.4. Идентификаторы и зарезервированные слова

Идентификатор – это просто имя. В JavaScript идентификаторы выступают в качестве имен переменных и функций, а также меток некоторых циклов. Идентификаторы в JavaScript должны начинаться с буквы, с символа подчеркивания (_) или знака доллара ($). Далее могут следовать любые буквы, цифры, символы подчеркивания или знаки доллара. (Цифра не может быть первым символом, так как тогда интерпретатору трудно будет отличать идентификаторы от чисел.) Примеры допустимых идентификаторов:

```
i
my_variable_name
v13
_dummy
$str
```

Для совместимости и простоты редактирования для составления идентификаторов обычно используются только символы ASCII и цифры. Однако JavaScript допускает возможность использования в идентификаторах букв и цифр из полного набора символов Юникода. (Технически стандарт ECMAScript также допускает наличие в идентификаторах символов Юникода из категорий Mn, Mc и Pc при условии, что они не являются первыми символами идентификаторов.) Это позволяет программистам давать переменным имена на своих родных языках и использовать в них математические символы:

```
var sí = true;
var π = 3.14;
```

Подобно другим языкам программирования, JavaScript резервирует некоторые идентификаторы. Эти «зарезервированные слова» не могут использоваться в качестве обычных идентификаторов. Они перечислены ниже.

2.4.1. Зарезервированные слова

JavaScript резервирует ряд идентификаторов, которые играют роль ключевых слов самого языка. Эти ключевые слова не могут служить идентификаторами в программах:

```
break       delete      function    return      typeof
case        do          if          switch      var
catch       else        in          this        void
continue    false       instanceof  throw       while
debugger    finally     new         true        with
default     for         null        try
```

JavaScript также резервирует некоторые ключевые слова, которые в настоящее время не являются частью языка, но которые могут войти в его состав в будущих версиях. Стандарт ECMAScript 5 резервирует следующие слова:

`class`
`const`
`enum`
`export`
`extends`
`import`
`super`

Ниже приводится дополнительный список слов, которые допустимы в обычном программном коде JavaScript и являются зарезервированными в строгом режиме:

`implements`
`interface`
`let`
`package`
`private`
`protected`
`public`
`static`
`yield`

В строгом режиме также вводится ограничение на использование следующих идентификаторов. Они не являются зарезервированными словами в полном понимании, но они не могут использоваться в качестве имен переменных, функций или параметров:

`arguments`
`eval`

Стандарт ECMAScript 3 резервирует все ключевые слова языка Java, и, хотя это требование было ослаблено в стандарте ECMAScript 5, тем не менее следует избегать использования этих идентификаторов, если необходимо обеспечить работоспособность JavaScript-кода при использовании реализаций JavaScript, соответствующих стандарту ECMAScript 3:

`abstract`
`boolean`
`byte`
`char`
`class`
`const`
`double`
`enum`
`export`
`extends`
`final`
`float`
`goto`
`implements`
`import`
`int`
`interface`
`long`
`native`
`package`
`private`
`protected`
`public`
`short`
`static`
`super`
`synchronized`
`throws`
`transient`
`volatile`

В языке JavaScript имеется множество предопределенных глобальных переменных и функций, имена которых не следует использовать для создания своих собственных переменных и функций:

`arguments`
`Array`
`Boolean`
`Date`
`decodeURI`
`decodeURIComponent`
`encodeURI`
`encodeURIComponent`
`Error`
`eval`
`EvalError`
`Function`
`Infinity`
`isFinite`
`isNaN`
`JSON`
`Math`
`NaN`
`Number`
`Object`
`parseFloat`
`parseInt`
`RangeError`
`ReferenceError`
`RegExp`
`String`
`SyntaxError`
`TypeError`
`undefined`
`URIError`

Имейте в виду, что конкретные реализации JavaScript могут содержать свои предопределенные глобальные переменные и функции. Кроме того, каждая конкретная платформа JavaScript (клиентская, серверная и прочие) может иметь свой собственный список глобальных свойств. В описании объекта `Window` в четвертой части книги приводится список глобальных переменных и функций, определяемых реализацией клиентского JavaScript.

2.5. Необязательные точки с запятой

Как и в других языках программирования, для отделения инструкций (глава 5) друг от друга в языке JavaScript используется точка с запятой (;). Использование точек с запятой имеет важное значение для ясного выражения намерений программиста: без этого разделителя по ошибке можно принять конец одной инструкции за начало следующей и наоборот. Обычно в JavaScript точку с запятой между инструкциями можно не ставить, если они находятся в разных строках. (Точку с запятой можно также опустить в конце программы или если следующей лексемой в программе является закрывающая фигурная скобка }.) Многие программисты на JavaScript используют точки с запятой для явного обозначения концов инструкций (этот же прием используется в примерах для этой книги), даже если в этом нет необходимости. Другие опускают точки с запятой везде, где только возможно, используя их лишь в некоторых ситуациях, где они совершенно необходимы. Прежде чем выбрать тот или иной стиль, вам необходимо познакомиться с некоторыми особенностями использования точек с запятой в JavaScript.

Взгляните на следующий фрагмент. Поскольку две инструкции находятся в разных строках, первую точку с запятой можно опустить:

```
a = 3;
b = 4;
```

Однако если эти инструкции записать, как показано ниже, первая точка с запятой становится обязательной:

```
a = 3; b = 4;
```

Обратите внимание, что в JavaScript не все разрывы строк интерпретируются как точка с запятой: разрывы строк обычно интерпретируются как точки с запятой, только когда интерпретатор оказывается неспособен выполнить синтаксический анализ программного кода без точек с запятой. Если говорить более формально (с двумя исключениями, описываемыми ниже), JavaScript интерпретирует разрыв строки как точку с запятой, если следующий непробельный символ не может быть интерпретирован как продолжение текущей инструкции. Взгляните на следующий фрагмент:

```
var a
a
=
3
console.log(a)
```

JavaScript интерпретирует этот программный код, как показано ниже:

```
var a; a = 3; console.log(a);
```

Интерпретатор JavaScript будет интерпретировать первый разрыв строки как точку с запятой, потому что он не сможет проанализировать фрагмент `var a a` без точки с запятой. Второй идентификатор `a` можно было бы интерпретировать как инструкцию `a;`, но JavaScript не будет воспринимать второй разрыв строки как точку с запятой, потому что он сможет продолжить синтаксический анализ и получить более длинную инструкцию `a = 3;`.

Эти правила интерпретации разрывов строк могут приводить к странным, на первый взгляд, ситуациям. Следующий фрагмент выглядит как две отдельные инструкции, отделенные символом перевода строки:

```
var y = x + f
(a+b).toString()
```

Однако круглые скобки во второй строке могут быть интерпретированы как вызов функции `f` из первой строки, и JavaScript будет интерпретировать этот фрагмент, как показано ниже:

```
var y = x + f(a+b).toString();
```

Вероятнее всего, программист вкладывал в этот фрагмент совсем иной смысл. Чтобы этот фрагмент интерпретировался как две отдельные инструкции, в данном случае необходимо использовать точку с запятой.

В целом, если инструкция начинается с символа (, [, /, + или -, есть вероятность, что она будет воспринята интерпретатором как продолжение предыдущей инструкции. Инструкции, начинающиеся с символов /, + и -, редко встречаются на практике, но инструкции, начинающиеся с символов (и [, встречаются достаточно часто, по крайней мере, при использовании некоторых стилей программирования на JavaScript. Некоторые программисты любят вставлять защитную точку с запятой в начало каждой такой инструкции, чтобы обеспечить корректную ее работу, даже если предыдущая инструкция будет изменена и ранее имевшаяся завершающая точка с запятой исчезнет:

```
var x = 0                         // Здесь точка с запятой опущена
;[x,x+1,x+2].forEach(console.log) // Защитная ; обеспечивает обособленность
                                  // этой инструкции
```

Из общего правила, согласно которому интерпретатор JavaScript воспринимает разрывы строк как точки с запятой, когда он не может интерпретировать вторую строку как продолжение инструкции в первой строке, имеется два исключения. Первое исключение связано с инструкциями `return`, `break` и `continue` (глава 5). Эти инструкции часто используются отдельно, но иногда вслед за ними указываются идентификаторы или выражения. Если разрыв строки находится сразу за любым из этих слов (перед любой другой лексемой), JavaScript всегда будет интерпретировать этот разрыв строки как точку с запятой. Например, если записать:

```
return
true;
```

интерпретатор JavaScript предположит, что программист имеет в виду следующее:

```
return; true;
```

Хотя на самом деле программист, видимо, хотел написать:

```
return true;
```

Это означает, что вы не должны вставлять разрыв строки между ключевым словом `return`, `break` или `continue` и выражением, следующим за ним. Если вставить разрыв строки в таком месте, программный код, скорее всего, будет порождать ошибку во время выполнения, которую будет сложно отыскать во время отладки.

Второе исключение связано с операторами `++` и `--` (раздел 4.8). Эти операторы могут быть префиксными, т. е. располагаться перед выражением, и постфиксными, т. е. располагаться после выражения. Если вам потребуется использовать любой из этих операторов в постфиксной форме записи, он должен находиться в той же строке, что и выражение, к которому применяется этот оператор. В противном случае разрыв строки будет интерпретироваться как точка с запятой, а оператор `++` или `--` будет интерпретироваться как префиксный оператор, применяемый к выражению, следующему далее. Например, взгляните на следующий фрагмент:

```
x
++
y
```

Он будет интерпретирован как `x; ++y;`, а не как `x++; y`.

3

Типы данных, значения и переменные

В процессе работы компьютерные программы манипулируют *значениями*, такими как число 3,14 или текст «Hello World». Типы значений, которые могут быть представлены и обработаны в языке программирования, известны как *типы данных*, и одной из наиболее фундаментальных характеристик любого языка программирования является поддерживаемый им набор типов данных. Когда в программе необходимо сохранить значение, чтобы использовать его позже, это значение присваивается (или сохраняется в) *переменной*. Переменная определяет символическое имя для значения и обеспечивает возможность получить это значение по имени. Принцип действия переменных является еще одной фундаментальной характеристикой любого языка программирования. В этой главе рассматриваются типы, значения и переменные в языке JavaScript. В этих вводных абзацах дается только краткий обзор, и в процессе их чтения вам, возможно, окажется полезным возвращаться к разделу 1.1. Более полное обсуждение этих тем вы найдете в последующих разделах.

Типы данных в JavaScript можно разделить на две категории: *простые типы* и *объекты*. К категории простых типов в языке JavaScript относятся числа, текстовые строки (которые обычно называют просто *строками*) и логические (или *булевы*) значения. Значительная часть этой главы посвящена подробному описанию числового (раздел 3.1) и строкового (раздел 3.2) типов данных. Логический тип рассматривается в разделе 3.3.

Специальные значения `null` и `undefined` являются элементарными значениями, но они не относятся ни к числам, ни к строкам, ни к логическим значениям. Каждое из них определяет только одно значение своего собственного специального типа. Подробнее о значениях `null` и `undefined` рассказывается в разделе 3.4.

Любое значение в языке JavaScript, не являющееся числом, строкой, логическим значением или специальным значением `null` или `undefined`, является объектом. Объект (т. е. член *объектного* типа данных) представляет собой коллекцию *свойств*, каждое из которых имеет имя и значение (либо простого типа, такое как число или строка, либо объектного). В разделе 3.5 мы рассмотрим один специальный объект, *глобальный объект*, но более подробно объекты обсуждаются в главе 6.

Обычный объект JavaScript представляет собой неупорядоченную коллекцию именованных значений. Кроме того, в JavaScript имеется объект специального типа, известный как *массив*, представляющий упорядоченную коллекцию пронумерованных значений. Для работы с массивами в языке JavaScript имеются специальные синтаксические конструкции. Кроме того, массивы ведут себя несколько иначе, чем обычные объекты. Подробнее о массивах будет рассказываться в главе 7.

В JavaScript определен еще один специальный тип объекта, известный как *функция*. Функция – это объект, с которым связан выполняемый код. Функция может *вызываться* для выполнения определенной операции и возвращать вычисленное значение. Подобно массивам, функции ведут себя не так, как другие виды объектов, и в JavaScript определен специальный синтаксис для работы с ними. Одна из важнейших особенностей функций в JavaScript состоит в том, что они являются самыми настоящими значениями, и программы JavaScript могут манипулировать ими, как обычными объектами. Подробнее о функциях рассказывается в главе 8.

Функции, которые пишутся для инициализации вновь создаваемых объектов (с оператором `new`), называются *конструкторами*. Каждый конструктор определяет *класс* объектов – множество объектов, инициализируемых этим конструктором. Классы можно представлять как подтипы объектного типа. В дополнение к классам `Array` и `Function` в базовом языке JavaScript определены еще три полезных класса. Класс `Date` определяет объекты, представляющие даты. Класс `RegExp` определяет объекты, представляющие регулярные выражения (мощный инструмент сопоставления с шаблоном, описываемый в главе 10). А класс `Error` определяет объекты, представляющие синтаксические ошибки и ошибки времени выполнения, которые могут возникать в программах на языке JavaScript. Имеется возможность определять собственные классы объектов, объявляя соответствующие функции-конструкторы. Подробнее об этом рассказывается в главе 9.

Интерпретатор JavaScript автоматически выполняет сборку мусора в памяти. Это означает, что программа может создавать объекты по мере необходимости, но программисту нет необходимости беспокоиться об уничтожении этих объектов и освобождении занимаемой ими памяти. Когда объект выходит за пределы области видимости (т. е. когда программа утрачивает возможность доступа к этому объекту) и интерпретатор обнаруживает, что данный объект никогда больше не сможет использоваться, он автоматически освобождает занимаемую им память.

JavaScript – это объектно-ориентированный язык программирования. В общих чертах это означает, что вместо глобальных функций для обработки значений различных типов типы сами могут определять методы для обработки значений. Например, чтобы отсортировать элементы массива `a`, необязательно передавать массив `a` функции `sort()`. Вместо этого можно просто вызвать метод `sort()` массива `a`:

```
a.sort();      // Объектно-ориентированная версия вызова sort(a).
```

Порядок определения методов описывается в главе 9. С технической точки зрения в языке JavaScript только объекты могут иметь методы. Однако числа, строки и логические значения ведут себя так, как если бы они обладали методами (данная особенность описывается в разделе 3.6). Значения `null` и `undefined` являются единственными в языке JavaScript, которые не имеют методов.

Типы данных в языке JavaScript можно разделить на простые и объектные. Их также можно разделить на типы с методами и типы без методов. Кроме того, типы можно характеризовать как *изменяемые* и *неизменяемые*. Значение изменяемого типа можно изменить. Объекты и массивы относятся к изменяемым типам: программа на языке JavaScript может изменять значения свойств объектов и элементов массивов. Числа, логические значения, `null` и `undefined` являются неизменяемыми – не имеет даже смысла говорить об изменчивости, например, значения числа. Строки можно представить себе как массивы символов, отчего можно счесть, что они являются изменяемыми. Однако строки в JavaScript являются неизменяемыми: строки предусматривают возможность обращения к символам по числовым индексам, но в JavaScript отсутствует возможность изменить существующую текстовую строку. Различия между изменяемыми и неизменяемыми значениями будут рассматриваться ниже, в разделе 3.7.

В языке JavaScript значения достаточно свободно могут быть преобразованы из одного типа в другой. Например, если программа ожидает получить строку, а вы передаете ей число, интерпретатор автоматически преобразует число в строку. Если вы укажете нелогическое значение там, где ожидается логическое, интерпретатор автоматически выполнит соответствующее преобразование. Правила преобразований описываются в разделе 3.8. Свобода преобразований типов значений в JavaScript затрагивает и понятие равенства, и оператор `==` проверки на равенство выполняет преобразование типов, как описывается в разделе 3.8.1.

Переменные в JavaScript не имеют типа: переменной может быть присвоено значение любого типа и позднее этой же переменной может быть присвоено значение другого типа. *Объявление* переменных выполняется с помощью ключевого слова `var`. В языке JavaScript используются лексические области видимости. Переменные, объявленные за пределами функции, являются *глобальными переменными* и доступны из любой точки программы. Переменные, объявленные внутри функции, находятся в области видимости функции и доступны только внутри этой функции. Порядок объявления переменных и их видимость обсуждаются в разделах 3.9 и 3.10.

3.1. Числа

В отличие от многих языков программирования, в JavaScript не делается различий между целыми и вещественными значениями. Все числа в JavaScript представляются вещественными значениями (с плавающей точкой). Для представления чисел в JavaScript используется 64-битный формат, определяемый стандартом IEEE 754.[1] Этот формат способен представлять числа в диапазоне от $\pm 1{,}7976931348623157 \times 10^{308}$ до $\pm 5 \times 10^{-324}$.

Формат представления вещественных чисел в JavaScript позволяет точно представлять все целые числа от -9007199254740992 (-2^{53}) до 9007199254740992 (2^{53}) включительно. Для целых значений вне этого диапазона может теряться точность в младших разрядах. Следует отметить, что некоторые операции в JavaScript

---

[1] Этот формат знаком программистам на языке Java как тип `double`. Этот же формат используется для представления чисел типа `double` в большинстве современных реализаций языков C и C++.

(такие как обращение к элементам массива по индексам и битовые операции, описываемые в главе 4) выполняются с 32-разрядными целыми значениями.

Число, находящееся непосредственно в программе на языке JavaScript, называется *числовым литералом*. JavaScript поддерживает числовые литералы нескольких форматов, описанных в последующих разделах. Обратите внимание, что любому числовому литералу может предшествовать знак «минус» (-), делающий числа отрицательными. Однако фактически минус представляет собой унарный оператор смены знака (см. главу 4), не являющийся частью синтаксиса числовых литералов.

3.1.1. Целые литералы

В JavaScript целые десятичные числа записываются как последовательность цифр. Например:

```
0
3
10000000
```

Помимо десятичных целых литералов JavaScript распознает шестнадцатеричные значения (по основанию 16). Шестнадцатеричные литералы начинаются с последовательности символов «0x» или «0X», за которой следует строка шестнадцатеричных цифр. Шестнадцатеричная цифра – это одна из цифр от 0 до 9 или букв от a (или A) до f (или F), представляющих значения от 10 до 15. Ниже приводятся примеры шестнадцатеричных целых литералов:

```
0xff          // 15*16 + 15 = 255 (по основанию 10)
0xCAFE911
```

Хотя стандарт ECMAScript не поддерживает представление целых литералов в восьмеричном формате (по основанию 8), некоторые реализации JavaScript допускают подобную возможность. Восьмеричный литерал начинается с цифры 0, за которой могут следовать цифры от 0 до 7. Например:

```
0377          // 3*64 + 7*8 + 7 = 255 (по основанию 10)
```

Поскольку некоторые реализации поддерживают восьмеричные литералы, а некоторые нет, никогда не следует писать целый литерал с ведущим нулем, ибо нельзя сказать наверняка, как он будет интерпретирован данной реализацией – как восьмеричное число или как десятичное. В строгом (strict) режиме, определяемом стандартом ECMAScript 5 (раздел 5.7.3), восьмеричные литералы явно запрещены.

3.1.2. Литералы вещественных чисел

Литералы вещественных чисел должны иметь десятичную точку – при определении таких литералов используется традиционный синтаксис вещественных чисел. Вещественное значение представляется как целая часть числа, за которой следуют десятичная точка и дробная часть числа.

Литералы вещественных чисел могут также представляться в экспоненциальной нотации: вещественное число, за которым следует буква e (или E), а затем необязательный знак плюс или минус и целая экспонента. Такая форма записи обозначает вещественное число, умноженное на 10 в степени, определяемой значением экспоненты.

Ниже приводится более лаконичное определение синтаксиса:

```
[цифры][.цифры][(E|e)[(+|-)]цифры]
```

Например:

```
3.14
2345.789
.333333333333333333
6.02e23         // 6.02 × 10^23
1.4738223E-32   // 1.4738223 × 10^-32
```

3.1.3. Арифметические операции в JavaScript

Обработка чисел в языке JavaScript выполняется с помощью арифметических операторов. В число таких операторов входят: оператор сложения +, оператор вычитания -, оператор умножения *, оператор деления / и оператор деления по модулю % (возвращает остаток от деления). Полное описание этих и других операторов можно найти в главе 4.

Помимо этих простых арифметических операторов JavaScript поддерживает более сложные математические операции, с помощью функций и констант, доступных в виде свойств объекта `Math`:

```
Math.pow(2,53)          // => 9007199254740992: 2 в степени 53
Math.round(.6)          // => 1.0: округление до ближайшего целого
Math.ceil(.6)           // => 1.0: округление вверх
Math.floor(.6)          // => 0.0: округление вниз
Math.abs(-5)            // => 5: абсолютное значение
Math.max(x,y,z)         // Возвращает наибольший аргумент
Math.min(x,y,z)         // Возвращает наименьший аргумент
Math.random()           // Псевдослучайное число x, где 0 <= x < 1.0
Math.PI                 // π: длина окружности / диаметр
Math.E                  // e: Основание натурального логарифма
Math.sqrt(3)            // Корень квадратный из 3
Math.pow(3, 1/3)        // Корень кубический из 3
Math.sin(0)             // Тригонометрия: имеются также Math.cos, Math.atan и другие.
Math.log(10)            // Натуральный логарифм 10
Math.log(100)/Math.LN10 // Логарифм 100 по основанию 10 (десятичный)
Math.log(512)/Math.LN2  // Логарифм 512 по основанию 2
Math.exp(3)             // Math.E в кубе
```

Полный перечень всех математических функций, поддерживаемых языком JavaScript, можно найти в справочном разделе с описанием объекта `Math`.

Арифметические операции в JavaScript не возбуждают ошибку в случае переполнения, потери значащих разрядов или деления на ноль. Если результат арифметической операции окажется больше самого большого представимого значения (переполнение), возвращается специальное значение «бесконечность», которое в JavaScript обозначается как `Infinity`. Аналогично, если абсолютное значение отрицательного результата окажется больше самого большого представимого значения, возвращается значение «отрицательная бесконечность», которое обозначается как `-Infinity`. Эти специальные значения, обозначающие бесконечность, ведут себя именно так, как и следовало ожидать: сложение, вычитание, умножение или деление бесконечности на любое значение дают в результате бесконечность (возможно, с обратным знаком).

Потеря значащих разрядов происходит, когда результат арифметической операции оказывается ближе к нулю, чем минимально возможное значение. В этом случае возвращается число 0. Если потеря значащих разрядов происходит в отрицательном результате, возвращается специальное значение, известное как «отрицательный ноль». Это специальное значение практически ничем не отличается от обычного нуля, и у программистов на JavaScript редко возникает необходимость выделять его.

Деление на ноль не считается ошибкой в JavaScript: в этом случае просто возвращается бесконечность или отрицательная бесконечность. Однако есть одно исключение: операция деления нуля на ноль не имеет четко определенного значения, поэтому в качестве результата такой операции возвращается специальное значение «не число» (not-a-number), которое обозначается как `NaN`. Значение `NaN` возвращается также при попытке разделить бесконечность на бесконечность, извлечь квадратный корень из отрицательного числа или выполнить арифметическую операцию с нечисловыми операндами, которые не могут быть преобразованы в числа.

В JavaScript имеются предопределенные глобальные переменные `Infinity` и `NaN`, хранящие значения положительной бесконечности и «не число». В стандарте ECMAScript 3 эти переменные доступны для чтения/записи и могут изменяться в программах. Стандарт ECMAScript 5 исправляет эту оплошность и требует, чтобы эти переменные были доступны только для чтения. Объект `Number` предоставляет альтернативные представления некоторых значений, доступные только для чтения даже в ECMAScript 3. Например:

```
Infinity                   // Переменная, доступная для чтения/записи,
                           // инициализированная значением Infinity.
Number.POSITIVE_INFINITY   // То же значение, доступное только для чтения.
1/0                        // То же самое значение.
Number.MAX_VALUE + 1       // Это выражение также возвращает Infinity.

Number.NEGATIVE_INFINITY   // Возвращают отрицательную бесконечность.
-Infinity
-1/0
-Number.MAX_VALUE - 1

NaN                        // Переменная, доступная для чтения/записи,
                           // инициализированная значением NaN.
Number.NaN                 // Свойство, доступное только для чтения, с тем же значением.
0/0                        // Возвращает NaN.

Number.MIN_VALUE/2         // Потеря значащих разрядов: возвращает 0
-Number.MIN_VALUE/2        // Отрицательный ноль
-1/Infinity                // Также отрицательный ноль
-0
```

Значение «не число» в JavaScript обладает одной необычной особенностью: операция проверки на равенство всегда возвращает отрицательный результат, даже если сравнить его с самим собой. Это означает, что нельзя использовать проверку `x == NaN`, чтобы определить, является значение переменной `x` значением `NaN`. Вместо этого следует выполнять проверку `x != x`. Эта проверка вернет `true` тогда и только тогда, когда `x` имеет значение `NaN`. Аналогичную проверку можно выполнить с помощью функции `isNaN()`. Она возвращает `true`, если аргумент имеет значение `NaN` или если аргумент является нечисловым значением, таким как строка или объект. Родственная функция `isFinite()` возвращает `true`, если аргумент является числом, отличным от `NaN`, `Infinity` или `-Infinity`.

Отрицательный ноль также имеет свои характерные особенности. В операциях сравнения (даже в операции строгой проверки на равенство) он признается равным положительному нулю, что делает эти два значения практически неотличимыми, за исключением случаев, когда они выступают в роли делителей:

```
var zero = 0;        // Обычный ноль
var negz = -0;       // Отрицательный ноль
zero === negz        // => true: ноль и отрицательный ноль равны
1/zero === 1/negz    // => false: Infinity и -Infinity не равны
```

3.1.4. Двоичное представление вещественных чисел и ошибки округления

Вещественных чисел существует бесконечно много, но формат представления вещественных чисел в JavaScript позволяет точно выразить лишь ограниченное их количество (точнее, 18437736874454810627). Это значит, что при работе с вещественными числами в JavaScript представление числа часто будет являться округлением фактического числа.

Стандарт представления вещественных чисел IEEE-754, используемый в JavaScript (и практически во всех других современных языках программирования), определяет двоичный формат их представления, который может обеспечить точное представление таких дробных значений, как 1/2, 1/8 и 1/1024. К сожалению, чаще всего мы пользуемся десятичными дробями (особенно при выполнении финансовых расчетов), такими как 1/10, 1/100 и т. д. Двоичное представление вещественных чисел неспособно обеспечить точное представление таких простых чисел, как `0.1`.

Точность представления вещественных чисел в JavaScript достаточно высока и позволяет обеспечить очень близкое представление числа `0.1`. Но тот факт, что это число не может быть представлено точно, может приводить к проблемам. Взгляните на следующий фрагмент:

```
var x = .3 - .2;   // тридцать копеек минус двадцать копеек
var y = .2 - .1;   // двадцать копеек минус 10 копеек
x == y             // => false: получились два разных значения!
x == .1            // => false: .3-.2 не равно .1
y == .1            // => true: .2-.1 равно .1
```

Из-за ошибок округления разность между аппроксимациями чисел .3 и .2 оказалась не равной разности между аппроксимациями чисел .2 и .1. Важно понимать, что эта проблема не является чем-то характерным для JavaScript: она проявляется во всех языках программирования, где используется двоичное представление вещественных чисел. Кроме того, обратите внимание, что значения `x` и `y` в примере выше *очень* близки друг к другу и к истинному значению. Точность округления вполне приемлема для большинства применений: проблема возникает лишь при попытках проверить значения на равенство.

В будущих версиях JavaScript может появиться поддержка десятичных чисел, лишенная описанных недостатков, связанных с округлением. Но до тех пор для важных финансовых расчетов предпочтительнее будет использовать масштабируемые целые числа. Например, финансовые расчеты можно производить в копейках, а не в долях рублей.

3.1.5. Дата и время

В базовом языке JavaScript имеется конструктор `Date()` для создания объектов, представляющих дату и время. Эти объекты `Date` обладают методами для выполнения простых вычислений с участием дат. Объект `Date` не является фундаментальным типом данных, как числа. Этот раздел представляет собой краткое пособие по работе с датами. Полное описание можно найти в справочном разделе:

```
var then = new Date(2010, 0, 1);             // Первый день первого месяца 2010 года
var later = new Date(2010, 0, 1, 17, 10, 30); // Та же дата, в 17:10:30 локального времени
var now = new Date();                        // Текущие дата и время
var elapsed = now - then;                    // Разность дат: интервал в миллисекундах

later.getFullYear()                          // => 2010
later.getMonth()                             // => 0: счет месяцев начинается с нуля
later.getDate()                              // => 1: счет дней начинается с единицы
later.getDay()                               // => 5: день недели. 0 - воскр., 5 - пятн.
later.getHours()                             // => 17: 17 часов локального времени
later.getUTCHours()                          // часы по UTC; зависит от часового пояса
later.toString()                             // => "Fri Jan 01 2010 17:10:30 GMT+0300"
later.toUTCString()                          // => "Fri, 01 Jan 2010 14:10:30 GMT"
later.toLocaleDateString()                   // => "1 Январь 2010 г."
later.toLocaleTimeString()                   // => "17:10:30"
later.toISOString()                          // => "2010-01-01T14:10:30.000Z"
```

3.2. Текст

Строка – это неизменяемая, упорядоченная последовательность 16-битных значений, каждое из которых обычно представляет символ Юникода. Строки в JavaScript являются типом данных, используемым для представления текста. *Длина* строки – это количество 16-битных значений, содержащихся в ней. Нумерация символов в строках (и элементов в массивах) в языке JavaScript начинается с нуля: первое 16-битное значение находится в позиции 0, второе – в позиции 1 и т. д. *Пустая строка* – это строка, длина которой равна 0. В языке JavaScript нет специального типа для представления единственного элемента строки. Для представления единственного 16-битного значения просто используется строка с длиной, равной 1.

3.2.1. Строковые литералы

Чтобы включить литерал строки в JavaScript-программу, достаточно просто заключить символы строки в парные одинарные или двойные кавычки (' или "). Символы двойных кавычек могут содержаться в строках, ограниченных символами одинарных кавычек, а символы одинарных кавычек – в строках, ограниченных символами двойных кавычек. Ниже приводятся несколько примеров строковых литералов:

```
               // Это пустая строка: в ней ноль символов
'testing'
"3.14"
'name="myform"'
"Вы предпочитаете книги издательства O'Reilly, не правда ли?"
"В этом строковом литерале\nдве строки"
"π - это отношение длины окружности к ее диаметру"
```

В ECMAScript 3 строковые литералы должны записываться в одной строке программы и не могут разбиваться на две строки. Однако в ECMAScript 5 строковые литералы можно разбивать на несколько строк, заканчивая каждую строку, кроме последней, символом обратного слэша (\). Ни один из символов обратного слэша, как и следующие за ними символы перевода строки, не будут включены в строковый литерал. Чтобы включить в строковый литерал символ перевода строки, следует использовать последовательность символов `\n` (описывается ниже):

```
"две\nстроки"  // Строковый литерал, представляющий две строки
"одна\         // Одна строка, записанная в трех строках. Только в ECMAScript 5
 длинная\
 строка"
```

Символы, кодовые пункты и строки JavaScript

Для представления символов Юникода в языке JavaScript используется кодировка UTF-16, а строки JavaScript являются последовательностями 16-битных значений без знака. Большинство наиболее часто используемых символов Юникода (из «основной многоязыковой матрицы») имеют кодовые пункты, умещающиеся в 16 бит, и могут быть представлены единственным элементом строки. Символы Юникода, кодовые пункты которых не умещаются в 16 бит, кодируются в соответствии с правилами кодировки UTF-16 как последовательности (известные как «суррогатные пары») из двух 16-битных значений. Это означает, что строка JavaScript, имеющая длину, равную 2 (два 16-битных значения), может представлять единственный символ Юникода:

```
var p = "π"; // π - это 1 символ с 16-битным кодовым пунктом 0x03c0
var e = "𝑒"; // 𝑒 - это 1 символ с 17-битным кодовым пунктом 0x1d452
p.length     // => 1: p содержит единственный 16-битный элемент
e.length     // => 2: в кодировке UTF-16 символ 𝑒 определяется двумя
             // 16-битными значениями: "\ud835\udc52"
```

Различные строковые методы, имеющиеся в языке JavaScript, манипулируют 16-битными значениями, а не символами. Они не предусматривают возможность специальной интерпретации суррогатных пар, не выполняют нормализацию строк и даже не проверяют, является ли строка последовательностью символов в кодировке UTF-16.

Обратите внимание, что, ограничивая строку одинарными кавычками, необходимо проявлять осторожность в обращении с апострофами, употребляемыми в английском языке для обозначения притяжательного падежа и в сокращениях; как, например, в словах «can't» и «O'Reilly's». Поскольку апостроф и одиночная кавычка – это одно и то же, необходимо при помощи символа обратного слэша (\) «экранировать» апострофы, расположенные внутри одиночных кавычек (подробнее об этом – в следующем разделе).

Программы на клиентском JavaScript часто содержат строки HTML-кода, а HTML-код, в свою очередь, часто содержит строки JavaScript-кода. Как и в JavaScript, в языке HTML для ограничения строк применяются либо одинарные, либо двойные кавычки. Поэтому при объединении JavaScript- и HTML-кода есть смысл придерживаться одного «стиля» кавычек для JavaScript, а другого – для HTML. В следующем примере строка «Спасибо» в JavaScript-выражении заключена в одинарные кавычки, а само выражение, в свою очередь, заключено в двойные кавычки как значение HTML-атрибута обработчика событий:

```
<button onclick="alert('Спасибо')">Щелкни на мне</button>
```

3.2.2. Управляющие последовательности в строковых литералах

Символ обратного слэша (\) имеет специальное назначение в JavaScript-строках. Вместе с символами, следующими за ним, он обозначает символ, не представимый внутри строки другими способами. Например, `\n` – это *управляющая последовательность* (escape sequence), обозначающая символ перевода строки.

Другой пример, упомянутый выше, – это последовательность \', обозначающая символ одинарной кавычки. Эта управляющая последовательность необходима для включения символа одинарной кавычки в строковый литерал, заключенный в одинарные кавычки. Теперь становится понятно, почему мы называем эти последовательности управляющими – здесь символ обратного слэша позволяет управлять интерпретацией символа одинарной кавычки. Вместо того чтобы отмечать ею конец строки, мы используем ее как апостроф:

```
'You\'re right, it can\'t be a quote'
```

В табл. 3.1 перечислены управляющие последовательности JavaScript и обозначаемые ими символы. Две управляющие последовательности являются обобщенными; они могут применяться для представления любого символа путем указания кода символа из набора Latin-1 или Unicode в виде шестнадцатеричного числа. Например, последовательность `\xA9` обозначает символ копирайта, который в кодировке Latin-1 имеет шестнадцатеричный код `A9`. Аналогично управляющая последовательность, начинающаяся с символов `\u`, обозначает произвольный символ Юникода, заданный четырьмя шестнадцатеричными цифрами. Например, `\u03c0` обозначает символ π.

Если символ «\» предшествует любому символу, отличному от приведенных в табл. 3.1, обратный слэш просто игнорируется (хотя будущие версии могут, конечно, определять новые управляющие последовательности). Например, `\#` – это то же самое, что и `#`. Наконец, как отмечалось выше, стандарт ECMAScript 5 позволяет добавлять в многострочные строковые литералы символ обратного слэша перед разрывом строки.

Таблица 3.1. Управляющие последовательности JavaScript

Последовательность	Представляемый символ
\0	Символ NUL (\u0000)
\b	«Забой» (\u0008)
\t	Горизонтальная табуляция (\u0009)
\n	Перевод строки (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Одинарная кавычка (\u0027)
\\	Обратный слэш (\u005C)
\x*XX*	Символ Latin-1, заданный двумя шестнадцатеричными цифрами *XX*
\ux*XXXX*	Символ Unicode, заданный четырьмя шестнадцатеричными цифрами *XXXX*

3.2.3. Работа со строками

Одной из встроенных возможностей JavaScript является способность *конкатенировать* строки. Если оператор + применяется к числам, они складываются, а если к строкам – они объединяются, при этом вторая строка добавляется в конец первой. Например:

```
msg = "Hello, " + "world";  // Получается строка "Hello, world"
greeting = "Добро пожаловать на мою домашнюю страницу," + " " + name;
```

Для определения длины строки – количества содержащихся в ней 16-битных значений – используется свойство строки `length`. Например, длину строки `s` можно получить следующим образом:

```
s.length
```

Кроме того, в дополнение к свойству `length` строки имеют множество методов (как обычно, более полную информацию ищите в справочном разделе):

```
var s = "hello, world"      // Начнем с того же текста.
s.charAt(0)                 // => "h": первый символ.
s.charAt(s.length-1)        // => "d": последний символ.
s.substring(1,4)            // => "ell": 2-й, 3-й и 4-й символы.
s.slice(1,4)                // => "ell": то же самое
s.slice(-3)                 // => "rld": последние 3 символа
s.indexOf("l")              // => 2: позиция первого символа l.
s.lastIndexOf("l")          // => 10: позиция последнего символа l.
s.indexOf("l", 3)           // => 3: позиция первого символа "l", следующего
                            //      за 3 символом в строке
s.split(", ")               // => ["hello", "world"] разбивает на подстроки
s.replace("h", "H")         // => "Hello, world": замещает все вхождения подстроки
s.toUpperCase()             // => "HELLO, WORLD"
```

Не забывайте, что строки в JavaScript являются неизменяемыми. Такие методы, как `replace()` и `toUpperCase()` возвращают новые строки: они не изменяют строку, относительно которой были вызваны.

В стандарте ECMAScript 5 строки могут интерпретироваться как массивы, доступные только для чтения, и вместо использования метода `charAt()` к отдельным символам (16-битным значениям) строки можно обращаться с помощью индексов в квадратных скобках:

```
s = "hello, world";
s[0]          // => "h"
s[s.length-1] // => "d"
```

Веб-броузеры, основанные на движке Mozilla, такие как Firefox, уже давно предоставляют такую возможность. Большинство современных броузеров (заметным исключением из которых является IE) последовали за Mozilla еще до того, как эта особенность была утверждена в стандарте ECMAScript 5.

3.2.4. Сопоставление с шаблонами

В языке JavaScript определен конструктор `RegExp()`, предназначенный для создания объектов, представляющих текстовые шаблоны. Эти шаблоны описываются с помощью *регулярных выражений*, синтаксис которых был заимствован языком JavaScript из языка Perl. И строки, и объекты `RegExp` имеют методы, позволяющие выполнять операции сопоставления с шаблоном и поиска с заменой при помощи регулярных выражений.

`RegExp` не относится к числу фундаментальных типов данных языка JavaScript. Подобно объектам `Date`, они просто являются специализированной разновидностью объектов с удобным прикладным интерфейсом. Грамматика регулярных выражений и прикладной интерфейс отличаются повышенной сложностью. Они подробно описываются в главе 10. Однако поскольку объекты `RegExp` обладают широкими возможностями и часто используются на практике, мы коротко познакомимся с ними в этом разделе.

Несмотря на то что объекты `RegExp` не относятся к фундаментальным типам данных языка, они имеют синтаксис литералов и могут вставляться непосредственно в текст программы на языке JavaScript. Текст, заключенный в пару символов слэша, интерпретируется как литерал регулярного выражения. За вторым символом слэша из этой пары может следовать один или более символов, которые модифицируют поведение шаблона. Например:

```
/^HTML/             // Соответствует символам H T M L в начале строки
/[1-9][0-9]*/       // Соответствует цифре, кроме нуля, за которой следует любое число цифр
/\bjavascript\b/i   // Соответствует подстроке "javascript"
                    // как отдельному слову, учитывает регистр символов
```

Объекты `RegExp` обладают множеством полезных методов. Кроме того, строки также обладают методами, которые принимают объекты `RegExp` в виде аргументов. Например:

```
var text = "testing: 1, 2, 3"; // Образец текста
var pattern = /\d+/g           // Соответствует всем вхождениям одной или более цифр
pattern.test(text)             // => true: имеется совпадение
text.search(pattern)           // => 9: позиция первого совпадения
text.match(pattern)          // => ["1", "2", "3"]: массив всех совпадений
text.replace(pattern, "#");  // => "testing: #, #, #"
text.split(/\D+/);           // => ["","1","2","3"]: разбить по нецифровым символам
```

3.3. Логические значения

Логическое значение говорит об истинности или ложности чего-то. Логический тип данных имеет только два допустимых логических значения. Эти два значения представлены литералами `true` и `false`.

Логические значения обычно представляют собой результат операций сравнения, выполняемых в JavaScript-программах. Например:

```
a == 4
```

Это выражение проверяет, равно ли значение переменной `a` числу 4. Если да, результатом этого сравнения будет логическое значение `true`. Если значение переменной `a` не равно 4, результатом сравнения будет `false`.

Логические значения обычно используются в управляющих конструкциях JavaScript. Например, инструкция `if/else` в JavaScript выполняет одно действие, если логическое значение равно `true`, и другое действие, если `false`. Обычно сравнение, создающее логическое значение, непосредственно объединяется с инструкцией, в которой оно используется. Результат выглядит так:

```
if (a == 4)
    b = b + 1;
else
    a = a + 1;
```

Здесь выполняется проверка равенства значения переменной `a` числу 4. Если равно, к значению переменной `b` добавляется 1; в противном случае число 1 добавляется к значению переменной `a`.

Как будет говориться в разделе 3.8, любое значение в языке JavaScript может быть преобразовано в логическое значение. Следующие значения в результате такого преобразования дают логическое значение (и затем работают как) `false`:

```
undefined
null
0
-0
NaN
""              // пустая строка
```

Все остальные значения, включая все объекты (и массивы), при преобразовании дают в результате значение (и работают как) `true`. Значение `false` и шесть значений, которые при преобразовании приводятся к этому значению, иногда называют *ложными*, а все остальные – *истинными*. В любом контексте, когда интерпретатор JavaScript ожидает получить логическое значение, ложные значения интерпретируются как `false`, а истинные значения – как `true`.

В качестве примера предположим, что переменная `o` может хранить объект или значение `null`. В этом случае можно явно проверить значение переменной `o` на неравенство значению `null`, как показано ниже:

```
if (o !== null) ...
```

Оператор «не равно» `!==` сравнит переменную `o` со значением `null` и вернет в результате `true` или `false`. Однако вы можете опустить оператор сравнения и положиться на тот факт, что `null` является ложным значением, а объект – истинным:

```
if (o) ...
```

В первом случае тело инструкции `if` будет выполнено, только если значение переменной `o` не равно `null`. Во втором – ставится менее жесткое условие: тело инструкции `if` будет выполнено, только если `o` не содержит `false` или другое ложное значение (такое как `null` или `undefined`). Какая инструкция `if` больше подходит для вашей программы, зависит от того, какие значения могут присваиваться переменной `o`. Если в программе необходимо отличать значение `null` от `0` и `""`, то следует использовать явную операцию сравнения.

Логические значения имеют метод `toString()`, который можно использовать для преобразования этих значений в строки «true» или «false», но они не имеют других полезных методов. Несмотря на простоту прикладного интерфейса, в языке имеется три важных логических оператора.

Оператор `&&` выполняет логическую операцию И. Он возвращает истинное значение, только если оба операнда истинны – в противном случае он возвращает ложное значение. Оператор `||` выполняет логическую операцию ИЛИ: он возвращает истинное значение, если хотя бы один (или оба) из операндов является истинным, и ложное значение – если оба операнда являются ложными. Наконец, унарный оператор `!` выполняет логическую операцию НЕ: он возвращает значение `true` для ложного операнда и `false` – для истинного. Например:

```
if ((x == 0 && y == 0) || !(z == 0)) {
    // x и y содержат значение 0 или z не равна нулю
}
```

Полное описание этих операторов приводится в разделе 4.10.

3.4. Значения null и undefined

Ключевое слово `null` в языке JavaScript имеет специальное назначение и обычно используется для обозначения отсутствия значения. Оператор `typeof` для значения `null` возвращает строку «object», что говорит о том, что значение `null` является специальным «пустым» объектом. Однако на практике значение `null` обычно считается единственным членом собственного типа и может использоваться как признак отсутствия значения, такого как число, строка или объект. В большинстве других языков программирования имеются значения, аналогичные значению `null` в JavaScript: вам они могут быть известны как `null` или `nil`.

В языке JavaScript имеется еще одно значение, свидетельствующее об отсутствии значения. Значение `undefined`, указывающее на полное отсутствие какого-либо значения. Оно возвращается при обращении к переменной, которой никогда не присваивалось значение, а также к несуществующему свойству объекта или элементу массива. Кроме того, значение `undefined` возвращается функциями, не имеющими возвращаемого значения, и присваивается параметрам функций для аргументов, которые не были переданы при вызове. Идентификатор `undefined` является именем предопределенной глобальной переменной (а не ключевым словом, как `null`), которая инициализирована значением `undefined`. В ECMAScript 3 `undefined` является переменной, доступной для чтения/записи, которой можно присвоить любое другое значение. Эта проблема была исправлена в ECMAScript 5, и в реализациях JavaScript, соответствующих этому стандарту, переменная `undefined` доступна только для чтения. Оператор `typeof` для значения `undefined` возвращает строку «undefined», показывающую, что данное значение является единственным членом специального типа.

Несмотря на эти отличия, оба значения, `null` и `undefined`, являются признаком отсутствия значения и часто являются взаимозаменяемыми. Оператор равенства == считает их равными. (Чтобы отличать их в программе, можно использовать оператор идентичности ===.) Оба они являются ложными значениями – в логическом контексте они интерпретируются как значение `false`. Ни `null`, ни `undefined` не имеют каких-либо свойств или методов. На практике попытка использовать `.` или `[]`, чтобы обратиться к свойству или методу этих значений, вызывает ошибку `TypeError`.

Значение `undefined` можно рассматривать как признак неожиданного или ошибочного отсутствия какого-либо значения, а `null` – как признак обычного или вполне ожидаемого отсутствия значения. Если в программе потребуется присвоить одно из этих значений переменной или свойству или передать одно из этих значений функции, практически всегда предпочтительнее использовать значение `null`.

3.5. Глобальный объект

В разделах выше описывались простые типы данных и значения языка JavaScript. Объектные типы – объекты, массивы и функции – описываются в отдельных главах, далее в книге. Но существует один важный объект, с которым необходимо познакомиться сейчас. *Глобальный объект* – это обычный объект JavaScript, который играет очень важную роль: свойства этого объекта являются глобальными идентификаторами, доступными из любого места в программах на JavaScript. Когда выполняется запуск интерпретатора JavaScript (или когда веб-броузер загружает новую страницу), создается новый глобальный объект, в котором инициализируется начальный набор свойств, определяющих:

- глобальные свойства, такие как `undefined`, `Infinity` и `NaN`;
- глобальные функции, такие как `isNaN()`, `parseInt()` (раздел 3.8.2) и `eval()` (раздел 4.12);
- функции-конструкторы, такие как `Date()`, `RegExp()`, `String()`, `Object()` и `Array()` (раздел 3.8.2);
- глобальные объекты, такие как `Math` и `JSON` (раздел 6.9).

Имена первоначально устанавливаемых свойств глобального объекта не являются зарезервированными словами, но вы вполне можете считать их таковыми. Все эти свойства перечислены в разделе 2.4.1. Некоторые из глобальных свойств уже описывались в этой главе. Большинство других будут рассматриваться в разных разделах книги. Кроме того, их все можно отыскать по именам в справочном разделе по базовому JavaScript или в описании самого глобального объекта, под именем «Global». В клиентском JavaScript имеется объект `Window`, определяющий другие глобальные свойства, описание которых можно найти в справочном разделе по клиентскому JavaScript.

В программном коде верхнего уровня, т. е. в JavaScript-коде, который не является частью функции, сослаться на глобальный объект можно посредством ключевого слова `this`:

```
var global = this; // Определить глобальную переменную для ссылки на глобальный объект
```

В клиентском JavaScript роль глобального объекта для всего JavaScript-кода, содержащегося в соответствующем ему окне броузера, играет объект `Window`. Этот глобальный объект имеет свойство `window`, ссылающееся на сам объект, которое можно использовать вместо ключевого слова `this` для ссылки на глобальный объект. Объект `Window` определяет базовые глобальные свойства, а также дополнительные глобальные свойства, характерные для веб-броузеров и клиентского JavaScript.

При создании в глобальном объекте определяются все предопределенные глобальные значения JavaScript. Однако этот специальный объект может также хранить глобальные переменные программы. Если программа объявляет глобальную переменную, она становится свойством глобального объекта. Подробнее этот механизм описывается в разделе 3.10.2.

3.6. Объекты-обертки

Объекты в языке JavaScript являются составными значениями: они представляют собой коллекции свойств, или именованных значений. Обращение к свойствам мы будем выполнять с использованием точечной нотации. Свойства, значениями которых являются функции, мы будем называть *методами*. Чтобы вызвать метод `m` объекта `o`, следует использовать инструкцию `o.m()`.

Мы уже видели, что строки обладают свойствами и методами:

```
var s = "hello world!";                          // Строка
var word = s.substring(s.indexOf(" ")+1, s.length); // Использование свойств строки
```

Однако строки не являются объектами, так почему же они обладают свойствами? Всякий раз когда в программе предпринимается попытка обратиться к свойству строки `s`, интерпретатор JavaScript преобразует строковое значение в объект, как если бы был выполнен вызов `new String(s)`. Этот объект наследует (раздел 6.2.2) строковые методы и используется интерпретатором для доступа к свойствам. После обращения к свойству вновь созданный объект уничтожается. (От реализаций не требуется фактически создавать и уничтожать этот промежуточный объект, но они должны вести себя так, как если бы объект действительно создавался и уничтожался.)

Наличие методов у числовых и логических значений объясняется теми же причинами: при обращении к какому-либо методу создается временный объект вызовом конструктора `Number()` или `Boolean()`, после чего производится вызов метода этого объекта. Значения `null` и `undefined` не имеют объектов-оберток: любые попытки обратиться к свойствам этих значений будет вызывать ошибку `TypeError`.

Рассмотрим следующий фрагмент и подумаем, что происходит при его выполнении:

```
var s = "test"; // Начальное строковое значение.
s.len = 4;      // Установить его свойство.
var t = s.len;  // Теперь запросить значение свойства.
```

В начале этого фрагмента переменная `t` имеет значение `undefined`. Вторая строка создает временный объект `String`, устанавливает его свойство `len` равным 4 и затем уничтожает этот объект. Третья строка создает из оригинальной (неизмененной) строки новый объект `String` и пытается прочитать значение свойства `len`. Строки не имеют данного свойства, поэтому выражение возвращает значение `undefined`. Данный фрагмент показывает, что при попытке прочитать значение какого-либо свойства (или вызвать метод) строки числа и логические значения ведут себя подобно объектам. Но если попытаться установить значение свойства, эта попытка будет просто проигнорирована: изменение затронет только временный объект и не будет сохранено.

Временные объекты, которые создаются при обращении к свойству строки, числа или логического значения, называются *объектами-обертками*, и иногда может потребоваться отличать строки от объектов `String` или числа и логические значения от объектов `Number` и `Boolean`. Однако обычно объекты-обертки можно рассматривать просто как особенность реализации и вообще не думать о них. Вам достаточно будет знать, что строки, числа и логические значения отличаются от объектов тем, что их свойства доступны только для чтения и что вы не можете определять для них новые свойства.

Обратите внимание, что существует возможность (но в этом почти никогда нет необходимости или смысла) явно создавать объекты-обертки вызовом конструктора `String()`, `Number()` или `Boolean()`:

```
var s = "test", n = 1, b = true; // Строка, чсло и логическое значение.
var S = new String(s);           // Объект String
var N = new Number(n);           // Объект Number
var B = new Boolean(b);          // Объект Boolean
```

При необходимости интерпретатор JavaScript обычно автоматически преобразует объекты-обертки, т. е. объекты `S`, `N` и `B` в примере выше, в обертываемые ими простые значения, но они не всегда ведут себя точно так же, как значения `s`, `n` и `b`. Оператор равенства == считает равными значения и соответствующие им объекты-обертки, но оператор идентичности === отличает их. Оператор `typeof` также обнаруживает отличия между простыми значениями и их объектами-обертками.

3.7. Неизменяемые простые значения и ссылки на изменяемые объекты

Между простыми значениями (`undefined`, `null`, логическими значениями, числами и строками) и объектами (включая массивы и функции) в языке JavaScript имеются фундаментальные отличия. Простые значения являются неизменяемыми: простое значение невозможно изменить (или «трансформировать»). Это очевидно для чисел и логических значений – нет никакого смысла изменять значение числа. Однако для строк это менее очевидно. Поскольку строки являются массивами символов, вполне естественно было бы ожидать наличие возможности изменять символы в той или иной позиции в строке. В действительности JavaScript не позволяет сделать это, и все строковые методы, которые, на первый взгляд, возвращают измененную строку, на самом деле возвращают новое строковое значение. Например:

```
var s = "hello"; // Изначально имеется некоторый текст из строчных символов
s.toUpperCase(); // Вернет "HELLO", но значение s при этом не изменится
s                // => "hello": оригинальная строка не изменилась
```

Кроме того, величины простых типов сравниваются *по значению*: две величины считаются одинаковыми, если они имеют одно и то же значение. Для чисел, логических значений, `null` и `undefined` это выглядит очевидным: нет никакого другого способа сравнить их. Однако для строк это утверждение не выглядит таким очевидным. При сравнении двух строковых значений JavaScript считает их одинаковыми тогда и только тогда, когда они имеют одинаковую длину и содержат одинаковые символы в соответствующих позициях.

Объекты отличаются от простых типов. Во-первых, они являются *изменяемыми* – их значения можно изменять:

```
var o = { x:1 }; // Начальное значение объекта
o.x = 2;         // Изменить, изменив значение свойства
o.y = 3;         // Изменить, добавив новое свойство

var a = [1,2,3]  // Массивы также являются изменяемыми объектами
a[0] = 0;        // Изменить значение элемента массив
a[3] = 4;        // Добавить новый элемент
```

Объекты не сравниваются по значению: два объекта не считаются равными, даже если они будут иметь одинаковые наборы свойств с одинаковыми значениями. И два массива не считаются равными, даже если они имеют один и тот же набор элементов, следующих в том же порядке:

```
var o = {x:1}, p = {x:1};         // Два объекта с одинаковыми свойствами
o === p                           // => false: разные объекты не являются равными
var a = [], b = [];               // Два различных пустых массива
a === b                           // => false: различные массивы не являются равными
```

Чтобы подчеркнуть отличие от простых типов JavaScript, объекты иногда называют *ссылочными типами*. Если следовать этой терминологии, значениями объектов являются *ссылки*, и можно сказать, что объекты сравниваются *по ссылке*: значения двух объектов считаются равными тогда и только тогда, когда они *ссылаются* на один и тот же объект в памяти.

```
var a = [];      // Переменная a ссылается на пустой массив.
var b = a;       // Теперь b ссылается на тот же массив.
b[0] = 1;        // Изменение массива с помощью ссылки в переменной b.
a[0]             // => 1: изменение можно наблюдать в переменной a.
a === b          // => true: a и b ссылаются на один и тот же объект, поэтому они равны.
```

Как следует из примера выше, операция присваивания объекта (или массива) переменной фактически присваивает ссылку: она не создает новую копию объекта. Если в программе потребуется создать новую копию объекта или массива, необходимо будет явно скопировать свойства объекта или элементы массива. Следующий пример демонстрирует такое копирование с помощью цикла `for` (раздел 5.5.3):

```
var a = ['a','b','c'];            // Копируемый массив
var b = [];                       // Массив, куда выполняется копирование
for(var i = 0; i < a.length; i++) { // Для каждого элемента в массиве a[]
    b[i] = a[i];                  // Скопировать элемент a[] в b[]
}
```

Точно так же, если потребуется сравнить два отдельных объекта или массива, необходимо будет сравнить значения их свойств или элементов. Ниже приводится определение функции, сравнивающей два массива:

```
function equalArrays(a,b) {
    if (a.length != b.length) return false;  // Массивы разной длины не равны
        for(var i = 0; i < a.length; i++)    // Цикл по всем элементам
            if (a[i] !== b[i]) return false; // Если хоть один элемент
                                             // отличается, массивы не равны
        return true;                         // Иначе они равны
}
```

3.8. Преобразование типов

JavaScript может гибко преобразовывать один тип в другой. Мы уже могли убедиться в этом на примере логических значений: везде, где интерпретатор JavaScript ожидает получить логическое значение, можно указать значение любого типа и JavaScript автоматически выполнит необходимое преобразование. Одни значения («истинные» значения) преобразуются в значение `true`, а другие («ложные») – в `false`. То же относится и к другим типам: если интерпретатор ожидает получить строку, он автоматически преобразует любое другое значение в строку. Если интерпретатор ожидает получить число, он попробует преобразовать имеющееся значение в число (в случае невозможности такого преобразования будет получено значение `NaN`). Например:

```
10 + " objects"  // => "10 objects". Число 10 преобразуется в строку
"7" * "4"        // => 28: обе строки преобразуются в числа
var n = 1 - "x"; // => NaN: строка "x" не может быть преобразована в число
n + " objects"   // => "NaN objects": NaN преобразуется в строку "NaN"
```

В табл. 3.2 описывается, как в JavaScript выполняется преобразование значений из одного типа в другой. Жирным шрифтом в таблице выделены значения, соответствующие преобразованиям, которые могут преподносить сюрпризы. Пустые ячейки соответствуют ситуациям, когда преобразование не требуется и не выполняется.

Преобразования одного простого типа в другой, показанные в табл. 3.2, выполняются относительно просто. Преобразование в логический тип уже обсуждалось в разделе 3.3. Преобразование всех простых типов в строку четко определено. Преобразование в число выполняется немного сложнее. Строки, которые могут быть преобразованы в числа, преобразуются в числа. В строке допускается наличие пробельных символов в начале и в конце, но присутствие других непробельных символов, которые не могут быть частью числа, при преобразовании строки в число приводят к возврату значения `NaN`. Некоторые особенности преобразования значений в числа могут показаться странными: значение `true` преобразуется в число 1, а значение `false` и пустая строка `""` преобразуются в `0`.

Преобразование простых типов в объекты также выполняется достаточно просто: значения простых типов преобразуются в соответствующие объекты-обертки (раздел 3.6), как если бы вызывался конструктор `String()`, `Number()` или `Boolean()`.

Исключение составляют значения `null` и `undefined`: любая попытка использовать их в контексте, где требуется объект, вместо преобразования будет приводить к возбуждению исключения `TypeError`.

Преобразование объектов в простые типы выполняется значительно сложнее и является темой обсуждения раздела 3.8.3.

Таблица 3.2. Преобразование типов в JavaScript

Значение	**Преобразование в:**			
	Строку	**Число**	**Логическое значение**	**Объект**
`undefined`	`"undefined"`	`NaN`	`false`	***возбуждается ошибка TypeError***
`null`	`"null"`	`0`	`false`	***возбуждается ошибка TypeError***
`true`	`"true"`	`1`		`new Boolean(true)`
`false`	`"false"`	`0`		`new Boolean(false)`
`""` **(пустая строка)**		`0`	`false`	`new String("")`
`"1.2"` **(непустая строка, число)**		`1.2`	`true`	`new String("1.2")`
`"one"` **(не пустая строка, не число)**		`NaN`	`true`	`new String("one")`
`0`	`"0"`		`false`	`new Number(0)`
`-0`	`"0"`		`false`	`new Number(-0)`
`NaN`	`"NaN"`		`false`	`new Number(NaN)`
`Infinity`	`"Infinity"`		`true`	`new Number(Infinity)`
`-Infinity`	`"-Infinity"`		`true`	`new Number(-Infinity)`
`1` **(конечное, ненулевое)**	`"1"`		`true`	`new Number(1)`
`{}` **(любой объект)**	*см. разд. 3.8.3*	*см. раздел 3.8.3*	`true`	
`[]` **(пустой массив)**	`""`	`0`	`true`	
`[9]` **(1 числовой элемент)**	`"9"`	`9`	`true`	
`['a']` **(любой другой массив)**	*используется метод* `join()`	`NaN`	`true`	
`function(){}` **(любая функция)**	*см. разд. 3.8.3*	`NaN`	`true`	

3.8.1. Преобразования и равенство

Благодаря гибкости преобразований типов в JavaScript оператор равенства `==` также гибко определяет равенство значений. Например, все следующие сравнения возвращают `true`:

```
null == undefined // Эти два значения считаются равными.
"0" == 0          // Перед сравнением строка преобразуется в число.
0 == false        // Перед сравнением логич. значение преобразуется в число.
"0" == false      // Перед сравнением оба операнда преобразуются в числа.
```

В разделе 4.9.1 четко описывается, какие преобразования выполняет оператор `==`, чтобы определить, являются ли два значения равными, и в этом же разделе описывается оператор идентичности `===`, который не выполняет никаких преобразований перед сравнением.

Имейте в виду, что возможность преобразования одного значения в другое не означает равенства этих двух значений. Если, например, в логическом контексте используется значение `undefined`, оно будет преобразовано в значение `false`. Но это не означает, что `undefined == false`. Операторы и инструкции JavaScript ожидают получить значения определенных типов и выполняют преобразования в эти типы. Инструкция `if` преобразует значение `undefined` в `false`, но оператор `==` никогда не пытается преобразовать свои операнды в логические значения.

3.8.2. Явные преобразования

Несмотря на то что многие преобразования типов JavaScript выполняет автоматически, иногда может оказаться необходимым выполнить преобразование явно или окажется предпочтительным выполнить явное преобразование, чтобы обеспечить ясность программного кода.

Простейший способ выполнить преобразование типа явно заключается в использовании функций `Boolean()`, `Number()`, `String()` и `Object()`. Мы уже видели, как эти функции используются в роли конструкторов объектов-оберток (раздел 3.6). При вызове без оператора `new` они действуют как функции преобразования и выполняют преобразования, перечисленные в табл. 3.2:

```
Number("3")    // => 3
String(false)  // => "false" или можно использовать false.toString()
Boolean([])    // => true
Object(3)      // => new Number(3)
```

Обратите внимание, что все значения, кроме `null` или `undefined`, имеют метод `toString()`, результатом которого обычно является то же значение, которое возвращается функцией `String()`. Кроме того, обратите внимание, что в табл. 3.2 отмечается, что при попытке преобразовать значение `null` или `undefined` в объект возбуждается ошибка `TypeError`. Функция `Object()` в этом случае не возбуждает исключение, вместо этого она просто возвращает новый пустой объект.

Определенные операторы в языке JavaScript неявно выполняют преобразования и иногда могут использоваться для преобразования типов. Если один из операндов оператора `+` является строкой, то другой операнд также преобразуется в строку. Унарный оператор `+` преобразует свой операнд в число. А унарный оператор `!` преобразует операнд в логическое значение и инвертирует его. Все это стало причиной появления следующих своеобразных способов преобразования типов, которые можно встретить на практике:

```
x + "" // То же, что и String(x)
+x     // То же, что и Number(x). Можно также встретить x-0
!!x    // То же, что и Boolean(x). Обратите внимание на два знака !
```

Форматирование и парсинг чисел являются наиболее типичными задачами, решаемыми компьютерными программами, и потому в JavaScript имеются специализированные функции и методы, обеспечивающие более полный контроль над преобразованиями чисел в строки и строк в числа.

Метод `toString()` класса `Number` принимает необязательный аргумент, определяющий основание системы счисления для преобразования. Если этот аргумент не определен, преобразование выполняется в десятичной системе счисления. Но вы можете производить преобразование в любой системе счисления (с основанием от 2 до 36). Например:

```
var n = 17;
binary_string = n.toString(2);      // Вернет "10001"
octal_string = "0" + n.toString(8); // Вернет "021"
hex_string = "0x" + n.toString(16); // Вернет "0x11"
```

При выполнении финансовых или научных расчетов может потребоваться обеспечить преобразование чисел в строки с точностью до определенного числа десятичных знаков или до определенного количества значащих разрядов или получать представление чисел в экспоненциальной форме. Для подобных преобразований чисел в строки класс `Number` определяет три метода. Метод `toFixed()` преобразует число в строку, позволяя указывать количество десятичных цифр после запятой. Он никогда не возвращает строки с экспоненциальным представлением чисел. Метод `toExponential()` преобразует число в строку в экспоненциальном представлении, когда перед запятой находится единственный знак, а после запятой следует указанное количество цифр (т. е. количество значащих цифр в строке получается на одну больше, чем было указано при вызове метода). Метод `toPrecision()` преобразует число в строку, учитывая количество заданных значащих разрядов. Если заданное количество значащих разрядов оказывается недостаточным для отображения всей целой части числа, преобразование выполняется в экспоненциальной форме. Обратите внимание, что все три метода округляют последние цифры или добавляют нули, если это необходимо. Взгляните на следующие примеры:

```
var n = 123456.789;
n.toFixed(0);          // "123457"
n.toFixed(2);          // "123456.79"
n.toFixed(5);          // "123456.78900"
n.toExponential(1);    // "1.2e+5"
n.toExponential(3);    // "1.235e+5"
n.toPrecision(4);      // "1.235e+5"
n.toPrecision(7);      // "123456.8"
n.toPrecision(10);     // "123456.7890"
```

Если передать строку функции преобразования `Number()`, она попытается разобрать эту строку как литерал целого или вещественного числа. Эта функция работает только с десятичными целыми числами и не допускает наличие в строке завершающих символов, не являющихся частью литерала числа. Функции `parseInt()` и `parseFloat()` (это глобальные функции, а не методы какого-либо класса) являются более гибкими. Функция `parseInt()` анализирует только целые числа, тогда как функция `parseFloat()` позволяет анализировать строки, представляющие и целые, и вещественные числа. Если строка начинается с последовательности «0x» или «0X», функция `parseInt()` интерпретирует ее как представление шестнадцатеричного числа.[1] Обе функции, `parseInt()` и `parseFloat()`, пропускают начальные пробельные символы, пытаются разобрать максимально возможное количество символов числа и игнорируют все, что следует за ними. Если первый непробельный символ строки не является частью допустимого числового литерала, эти функции возвращают значение NaN:

---

[1] Согласно стандарту ECMAScript 3 функция `parseInt()` может выполнять преобразование строки, начинающейся с символа «0» (но не «0x» или «0X»), в восьмеричное или десятичное число. Поскольку поведение функции четко не определено, следует избегать использования функции `parseInt()` для интерпретации строк, начинающихся с «0», или явно указывать основание системы счисления! В ECMAScript 5 функция `parseInt()` будет интерпретировать строки как восьмеричные числа, только если ей во втором аргументе явно указать основание 8 системы счисления.

```
parseInt("3 blind mice")   // => 3
parseFloat(" 3.14 meters") // => 3.14
parseInt("-12.34")         // => -12
parseInt("0xFF")           // => 255
parseInt("0xff")           // => 255
parseInt("-0XFF")          // => -255
parseFloat(".1")           // => 0.1
parseInt("0.1")            // => 0
parseInt(".1")             // => NaN: целые числа не могут начинаться с "."
parseFloat("$72.47");      // => NaN: числа не могут начинаться с "$"
```

Функция `parseInt()` принимает второй необязательный аргумент, определяющий основание системы счисления для разбираемого числа. Допустимыми являются значения от 2 до 36. Например:

```
parseInt("11", 2);         // => 3 (1*2 + 1)
parseInt("ff", 16);        // => 255 (15*16 + 15)
parseInt("zz", 36);        // => 1295 (35*36 + 35)
parseInt("077", 8);        // => 63 (7*8 + 7)
parseInt("077", 10);       // => 77 (7*10 + 7)
```

3.8.3. Преобразование объектов в простые значения

Преобразование объектов в логические значения выполняется очень просто: все объекты (включая массивы и функции) преобразуются в значение `true`. Это справедливо и для объектов-оберток: результатом вызова `new Boolean(false)` является объект, а не простое значение, поэтому он также преобразуется в значение `true`.

Преобразование объекта в строку и преобразование объекта в число выполняется вызовом соответствующего метода объекта. Все осложняется тем, что объекты в языке JavaScript имеют два разных метода для выполнения преобразований, а также наличием нескольких специальных случаев, описываемых ниже. Обратите внимание, что правила преобразования объектов в строки и числа, описываемые здесь, применяются только к объектам самого языка JavaScript. Объекты среды выполнения (например, определяемые веб-броузерами) могут предусматривать собственные алгоритмы преобразования в числа и строки.

Все объекты наследуют два метода преобразования. Первый из них называется `toString()`, он возвращает строковое представление объекта. По умолчанию метод `toString()` не возвращает ничего особенно интересного (хотя эта информация иногда может оказаться полезной, как будет показано в примере 6.4):

```
({x:1, y:2}).toString() // => "[object Object]"
```

Многие классы определяют более специализированные версии метода `toString()`. Например, метод `toString()` класса `Array` преобразует все элементы массива в строки и объединяет результаты в одну строку, вставляя запятые между ними. Метод `toString()` класса `Function` возвращает строковое представление функции, зависящее от реализации. На практике обычно реализации преобразуют пользовательские функции в строки с исходным программным кодом на языке JavaScript.

Класс `Date` определяет метод `toString()`, возвращающий строку с датой и временем в удобочитаемом формате (который может быть разобран средствами JavaScript). Класс `RegExp` определяет метод `toString()`, преобразующий объект `RegExp` в строку, которая выглядит как литерал регулярного выражения:

```
[1,2,3].toString()                  // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) {\n f(x);\n}"
/\d+/g.toString()                   // => "/\\d+/g"
new Date(2010,0,1).toString()       // => "Fri Jan 01 2010 00:00:00 GMT+0300"
```

Другая функция преобразования объектов называется `valueOf()`. Задача этого метода определена не так четко: предполагается, что он должен преобразовать объект в представляющее его простое значение, если такое значение существует. Объекты по своей природе являются составными значениями, и большинство объектов не могут быть представлены в виде единственного простого значения, поэтому по умолчанию метод `valueOf()` возвращает не простое значение, а сам объект. Классы-обертки определяют методы `valueOf()`, возвращающие обернутые простые значения. Массивы, функции и регулярные выражения наследуют метод по умолчанию. Вызов метода `valueOf()` экземпляров этих типов возвращает сам объект. Класс `Date` определяет метод `valueOf()`, возвращающий дату во внутреннем представлении: количество миллисекунд, прошедших с 1 января 1970 года:

```
var d = new Date(2010, 0, 1);       // 1 января 2010 года, (время Московское)
d.valueOf()                         // => 1262293200000
```

Теперь, разобравшись с методами `toString()` и `valueOf()`, можно перейти к обсуждению особенностей преобразования объектов в строки и в числа. Учтите, что существует несколько специальных случаев, когда JavaScript выполняет преобразование объектов в простые значения несколько иначе. Эти особые случаи рассматриваются в конце данного раздела.

Преобразование объектов в строку интерпретатор JavaScript выполняет в два этапа:

- Если объект имеет метод `toString()`, интерпретатор вызывает его. Если он возвращает простое значение, интерпретатор преобразует значение в строку (если оно не является строкой) и возвращает результат преобразования. Обратите внимание, что правила преобразований простых значений в строку четко определены для всех типов и перечислены в табл. 3.2.
- Если объект не имеет метода `toString()` или этот метод не возвращает простое значение, то интерпретатор проверяет наличие метода `valueOf()`. Если этот метод определен, интерпретатор вызывает его. Если он возвращает простое значение, интерпретатор преобразует это значение в строку (если оно не является строкой) и возвращает результат преобразования.
- В противном случае интерпретатор делает вывод, что ни `toString()`, ни `valueOf()` не позволяют получить простое значение и возбуждает исключение `TypeError`.

При преобразовании объекта в число интерпретатор выполняет те же действия, но первым пытается применить метод `valueOf()`:

- Если объект имеет метод `valueOf()`, возвращающий простое значение, интерпретатор преобразует (при необходимости) это значение в число и возвращает результат.

- Иначе, если объект имеет метод `toString()`, возвращающий простое значение, интерпретатор выполняет преобразование и возвращает полученное значение.
- В противном случае возбуждается исключение `TypeError`.

Описанный алгоритм преобразования объекта в число объясняет, почему пустой массив преобразуется в число 0, а массив с единственным элементом может быть преобразован в обычное число. Массивы наследуют по умолчанию метод `valueOf()`, который возвращает сам объект, а не простое значение, поэтому при преобразовании массива в число интерпретатор опирается на метод `toString()`. Пустые массивы преобразуются в пустую строку. А пустая строка преобразуется в число 0. Массив с единственным элементом преобразуется в ту же строку, что и единственный элемент массива. Если массив содержит единственное число, это число преобразуется в строку, а затем опять в число.

Оператор + в языке JavaScript выполняет сложение чисел и конкатенацию строк. Если какой-либо из его операндов является объектом, JavaScript преобразует объект, используя специальное преобразование объекта в простое значение вместо преобразования объекта в число, используемого другими арифметическими операторами. То же относится и к оператору равенства ==. Если выполняется сравнение объекта с простым значением, оператор выполнит преобразование объекта с использованием правил преобразования в простое значение.

Преобразование объектов в простые значения, используемое операторами + и ==, предусматривает особый подход для объектов `Date`. Класс `Date` является единственным типом данных в базовом JavaScript, который определяет осмысленные преобразования и в строку, и в число. Преобразование любого объекта, не являющегося датой, в простое значение основано на преобразовании в число (когда первым применяется метод `valueOf()`), тогда как для объектов типа `Date` используется преобразование в строку (первым применяется метод `toString()`). Однако преобразование выполняется не совсем так, как было описано выше: простое значение, возвращаемое методом `valueOf()` или `toString()`, используется непосредственно, без дополнительного преобразования в число или в строку.

Оператор < и другие операторы отношений выполняют преобразование объектов в простые значения подобно оператору ==, но не выделяя объекты `Date`: для любого объекта сначала предпринимается попытка применить метод `valueOf()`, а затем метод `toString()`. Любое простое значение, полученное таким способом, используется непосредственно, без дальнейшего преобразования в число или в строку.

+, ==, != и операторы отношений являются единственными, выполняющими специальное преобразование строки в простое значение. Другие операторы выполняют более явные преобразования в заданный тип и не предусматривают специальной обработки объектов `Date`. Оператор -, например, преобразует свои операнды в числа. Следующий фрагмент демонстрирует поведение операторов +, -, == и > при работе с объектами `Date`:

```
var now = new Date(); // Создать объект Date
typeof (now + 1)      // => "строка": + преобразует дату в строку
typeof (now - 1)      // => "число": - выполнит преобразование объекта в число
now == now.toString() // => true: неявное и явное преобразование в строку
now > (now -1)        // => true: > преобразует объект Date в число
```

3.9. Объявление переменных

Прежде чем использовать переменную в JavaScript, ее необходимо *объявить*. Переменные объявляются с помощью ключевого слова `var` следующим образом:

```
var i;
var sum;
```

Один раз использовав ключевое слово `var`, можно объявить несколько переменных:

```
var i, sum;
```

Объявление переменных можно совмещать с их инициализацией:

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

Если начальное значение в инструкции `var` не задано, то переменная объявляется, но ее начальное значение остается неопределенным (`undefined`), пока не будет изменено программой.

Обратите внимание, что инструкция `var` также может включаться в циклы `for` и `for/in` (о которых рассказывается в главе 5), что позволяет объявлять переменную цикла непосредственно в самом цикле. Например:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var p in o) console.log(p);
```

Если вы имеете опыт использования языков программирования со статическими типами данных, таких как С или Java, то можете заметить, что в объявлениях переменных в языке JavaScript отсутствует объявление типа. Переменные в языке JavaScript могут хранить значения любых типов. Например, в JavaScript допускается присвоить некоторой переменной число, а затем этой же переменной присвоить строку:

```
var i = 10;
i = "ten";
```

3.9.1 Повторные и опущенные объявления

С помощью инструкции `var` можно объявить одну и ту же переменную несколько раз. Если повторное объявление содержит инициализатор, то оно действует как обычная инструкция присваивания.

Если попытаться прочитать значение необъявленной переменной, JavaScript сгенерирует ошибку. В строгом режиме, предусмотренном стандартом ECMAScript 5 (раздел 5.7.3), ошибка также возбуждается при попытке присвоить значение необъявленной переменной. Однако исторически и при выполнении не в строгом режиме, если присвоить значение переменной, не объявленной с помощью инструкции `var`, то JavaScript создаст эту переменную как свойство глобального объекта, и она будет действовать практически так же (но с некоторыми отличиями, описываемыми в разделе 3.10.2), как корректно объявленная переменная. Это означает, что глобальные переменные можно не объявлять. Однако это считается дурной привычкой и может явиться источником ошибок, поэтому всегда старайтесь объявлять свои переменные с помощью var.

3.10. Область видимости переменной

Область видимости (scope) переменной – это та часть программы, для которой эта переменная определена. *Глобальная* переменная имеет глобальную область видимости – она определена для всей JavaScript-программы. В то же время переменные, объявленные внутри функции, определены только в ее теле. Они называются *локальными* и имеют локальную область видимости. Параметры функций также считаются локальными переменными, определенными только в теле этой функции.

Внутри тела функции локальная переменная имеет преимущество перед глобальной переменной с тем же именем. Если объявить локальную переменную или параметр функции с тем же именем, что у глобальной переменной, то фактически глобальная переменная будет скрыта:

```
var scope = "global";              // Объявление глобальной переменной
function checkscope() {
    var scope = "local";           // Объявление локальной переменной с тем же именем
    return scope;                  // Вернет локальное значение, а не глобальное
}
checkscope()                       // => "local"
```

Объявляя переменные с глобальной областью видимости, инструкцию var можно опустить, но при объявлении локальных переменных всегда следует использовать инструкцию var. Посмотрите, что получается, если этого не сделать:

```
scope = "global";                  // Объявление глобальной переменной, даже без var.

function checkscope2() {
    scope = "local";               // Ой! Мы изменили глобальную переменную.
    myscope = "local";             // Неявно объявляется новая глоб. переменная.
    return [scope, myscope];       // Вернуть два значения.
}
checkscope2()           // => ["local", "local"]: имеется побочный эффект!
scope                   // => "local": глобальная переменная изменилась.
myscope                 // => "local": нарушен порядок в глобальном пространстве имен.
```

Определения функций могут быть вложенными. Каждая функция имеет собственную локальную область видимости, поэтому может быть несколько вложенных уровней локальных областей видимости. Например:

```
var scope = "global scope";        // Глобальная переменная
function checkscope() {
    var scope = "local scope";     // Локальная переменная
    function nested() {
        var scope = "nested scope"; // Вложенная область видимости локальных переменных
        return scope;              // Вернет значение этой переменной scope
    }
    return nested();
}
checkscope()                       // => "nested scope"
```

3.10.1. Область видимости функции и подъем

В некоторых С-подобных языках программирования каждый блок программного кода внутри фигурных скобок имеет свою собственную область видимости, а переменные, объявленные внутри этих блоков, невидимы за их пределами. Эта особенность называется *областью видимости блока*, но она *не* поддерживается в языке JavaScript. Вместо этого в JavaScript используется такое понятие, как *область видимости функции*: переменные, объявленные внутри функции, доступны внутри функции, где они объявлены, а также внутри всех функций, вложенных в эту функцию.

В следующем фрагменте переменные `i`, `j` и `k` объявляются в разных местах, но все они имеют одну и ту же область видимости – все три переменные доступны из любого места в теле функции:

```
function test(o) {
    var i = 0;                      // i определена в теле всей функции
    if (typeof o == "object") {
        var j = 0;                  // j определена везде, не только в блоке
        for(var k=0; k < 10; k++) { // k определена везде, не только в цикле
            console.log(k);         // выведет числа от 0 до 9
        }
        console.log(k);             // k по-прежнему определена: выведет 10
    }
    console.log(j);                 // j определена, но может быть неинициализирована
}
```

Область видимости функции в языке JavaScript подразумевает, что все переменные, объявленные внутри функции, видимы *везде* в теле функции. Самое интересное, что переменные оказываются видимыми еще до того, как будут объявлены. Эта особенность JavaScript неофициально называется *подъемом*: программный код JavaScript ведет себя так, как если бы все объявления переменных внутри функции (без присваивания инициализирующих значений) «поднимались» в начало функции. Рассмотрим следующий фрагмент:

```
var scope = "global";
function f() {
    console.log(scope);  // Выведет "undefined", а не "global"
    var scope = "local"; // Инициализируется здесь, а определена везде
    console.log(scope);  // Выведет "local"
}
```

Можно было бы подумать, что первая инструкция внутри функции должна вывести слово «global», потому что инструкция `var` с объявлением локальной переменной еще не была выполнена. Однако вследствие действия правил области видимости функции выводится совсем другое значение. Локальная переменная определена во всем теле функции, а это означает, что глобальная переменная с тем же именем оказывается скрытой для всей функции. Хотя локальная переменная определена во всем теле функции, она остается неинициализированной до выполнения инструкции `var`. То есть функция выше эквивалентна реализации, приведенной ниже, в которой объявление переменной «поднято» в начало функции, а инициализация переменной выполняется там же, где и раньше:

```
function f() {
    var scope;           // Объявление локальной переменной в начале функции
    console.log(scope);  // Здесь она доступна, но имеет значение "undefined"
    scope = "local";     // Здесь она инициализируется и получает свое значение
    console.log(scope);  // А здесь она имеет ожидаемое значение
}
```

В языках программирования, где поддерживаются области видимости блоков, рекомендуется объявлять переменные как можно ближе к тому месту, где они используются, а область видимости делать как можно более узкой. Поскольку в JavaScript не поддерживаются области видимости блоков, некоторые программисты стремятся объявлять все переменные в начале функции, а не рядом с местом, где они используются. Такой подход позволяет более точно отражать истинную область видимости переменных в программном коде.

3.10.2. Переменные как свойства

При объявлении глобальной переменной в JavaScript в действительности создается свойство глобального объекта (раздел 3.5). Если глобальная переменная объявляется с помощью инструкции `var`, создается ненастраиваемое свойство (раздел 6.7), т. е. свойство, которое невозможно удалить с помощью оператора `delete`. Как уже отмечалось выше, если не используется строгий режим и необъявленной переменной присваивается некоторое значение, интерпретатор JavaScript автоматически создает глобальную переменную. Переменные, созданные таким способом, становятся обычными, настраиваемыми свойствами глобального объекта и могут быть удалены:

```
var truevar = 1;        // Правильно объявленная глобальная переменная, неудаляемая.
fakevar = 2;            // Создается удаляемое свойство глобального объекта.
this.fakevar2 = 3;      // То же самое.
delete truevar          // => false: переменная не была удалена
delete fakevar          // => true: переменная удалена
delete this.fakevar2    // => true: переменная удалена
```

Глобальные переменные в языке JavaScript являются свойствами глобального объекта, и такое положение вещей закреплено в спецификации ECMAScript. Это не относится к локальным переменным, однако локальные переменные можно представить как свойства объекта, ассоциированного с каждым вызовом функции. В спецификации ECMAScript 3 этот объект называется «объектом вызова» (call object), а в спецификации ECMAScript 5 он называется «записью с описанием окружения» (declarative environment record). Интерпретатор JavaScript позволяет ссылаться на глобальный объект с помощью ключевого слова `this`, но он не дает никакой возможности сослаться на объект, в котором хранятся локальные переменные. Истинная природа объектов, в которых хранятся локальные переменные, зависит от конкретной реализации и не должна заботить нас. Однако сам факт наличия объектов с локальными переменными имеет большое значение, и эта тема будет рассматриваться в следующем разделе.

3.10.3 Цепочки областей видимости

JavaScript – это язык программирования с *лексической областью видимости*: область видимости переменной распространяется на строки с исходным программным кодом, для которых определена переменная. Глобальные переменные определены для всей программы в целом. Локальные переменные определены для всей функции, в которой они объявлены, а также для любых функций, вложенных в эту функцию.

Если считать локальные переменные свойствами некоторого объекта, зависящего от реализации, то появляется возможность взглянуть на области видимости переменных с другой стороны. Каждый фрагмент программного кода на JavaScript (глобальный программный код или тело функции) имеет *цепочку областей видимости*, ассоциированную с ним. Эта цепочка областей видимости представляет собой список, или цепочку объектов, определяющих переменные, которые находятся «в области видимости» данного фрагмента программного кода. Когда интерпретатору требуется отыскать значение переменной `x` (этот процесс называется *разрешением переменной*), он начинает поиск с первого объекта в цепочке. Если этот объект имеет свойство с именем `x`, используется значение этого свойства. Если первый объект не имеет свойства с именем `x`, интерпретатор JavaScript продолжает поиск в следующем объекте в цепочке. Если второй объект не имеет свойства с именем `x`, интерпретатор переходит к следующему объекту и т. д. Если ни один из объектов в цепочке областей видимости не имеет свойства с именем `x`, то интерпретатор считает, что переменная `x` находится вне области видимости данного программного кода и возбуждает ошибку `ReferenceError`.

Для программного кода верхнего уровня (т. е. для программного кода за пределами каких-либо функций) цепочка областей видимости состоит из единственного, глобального объекта. Для невложенных функций цепочка областей видимости состоит из двух объектов. Первым является объект, определяющий параметры и локальные переменные функции, а вторым – глобальный объект. Для вложенных функций цепочка областей видимости может содержать три и более объектов. Важно понимать, как создаются цепочки этих объектов. Определение функции фактически сохраняет ее область видимости в цепочке. Когда эта функция вызывается, интерпретатор создает новый объект, хранящий локальные переменные, и добавляет его к имеющейся цепочке, образуя новую, более длинную цепочку, представляющую область видимости вызываемой функции. Ситуация становится еще более интересной для вложенных функций, потому что каждый раз, когда вызывается внешняя функция, внутренняя функция объявляется заново. Поскольку для каждого вызова внешней функции создается новая цепочка, вложенные функции будут немного отличаться при каждом определении – при каждом вызове внешней функции программный код вложенной функции будет одним и тем же, но цепочка областей видимости, ассоциированная с этим программным кодом, будет отличаться.

Такой взгляд на цепочку областей видимости будет полезен для понимания инструкции `with` (раздел 5.7.1) и чрезвычайно важен для понимания замыканий (раздел 8.6).

4

Выражения и операторы

Выражение – это фраза языка JavaScript, которая может быть *вычислена* интерпретатором для получения значения. Константа, встроенная в программу, является простейшей разновидностью выражений. Имя переменной также является простейшим выражением, в результате вычисления которого получается значение, присвоенное переменной. Более сложные выражения состоят из простых выражений. Выражение обращения к элементу массива, например, состоит из выражения, которое возвращает массив, за которым следуют открывающая квадратная скобка, выражение, возвращающее целое число, и закрывающая квадратная скобка. В результате вычисления этого более сложного выражения получается значение, хранящееся в элементе указанного массива с указанным индексом. Аналогично выражение вызова функции состоит из выражения, возвращающего объект функции и ноль или более дополнительных выражений, используемых в качестве аргументов функции.

Наиболее типичный способ конструирования сложных выражений из более простых выражений заключается в использовании *операторов*. Операторы объединяют значения своих операндов (обычно двух) некоторым способом и вычисляют новое значение. Простейшим примером может служить оператор умножения `*`. Выражение `x * y` вычисляется как произведение значений выражений `x` и `y`. Иногда для простоты мы говорим, что оператор *возвращает* значение вместо «вычисляет» значение.

Эта глава описывает все операторы JavaScript, а также выражения (такие как индексирование массивов и вызов функций), в которых не используются операторы. Те, кто знаком с другими языками программирования, использующими C-подобный синтаксис, заметят, что в JavaScript выражения и операторы имеют похожий синтаксис.

4.1. Первичные выражения

Простейшие выражения, известные как *первичные выражения*, являются самостоятельными выражениями – они не включают более простых выражений.

Первичными выражениями в языке JavaScript являются константы, или *литералы*, некоторые ключевые слова и ссылки на переменные.

Литералы и константы встраиваются непосредственно в программный код. Они выглядят, как показано ниже:

```
1.23        // Числовой литерал
"hello"     // Строковый литерал
/pattern/   // Литерал регулярного выражения
```

Синтаксис числовых литералов в JavaScript был описан в разделе 3.1. О строковых литералах рассказывалось в разделе 3.2. Синтаксис литералов регулярных выражений был представлен в разделе 3.2.4 и подробно будет описываться в главе 10.

Ниже приводятся некоторые из зарезервированных слов JavaScript, являющихся первичными выражениями:

```
true        // Возвращает логическое значение true
false       // Возвращает логическое значение false
null        // Возвращает значение null
this        // Возвращает "текущий" объект
```

Мы познакомились со значениями `true`, `false` и `null` в разделах 3.3 и 3.4. В отличие от других ключевых слов, `this` не является константой – в разных местах программы оно может возвращать разные значения. Ключевое слово `this` используется в объектно-ориентированном программировании. Внутри метода `this` возвращает объект, относительно которого был вызван метод. Дополнительные сведения о ключевом слове `this` можно найти в разделе 4.5, в главе 8 (особенно в разделе 8.2.2) и в главе 9.

Наконец, третьим типом первичных выражений являются ссылки на переменные:

```
i           // Возвращает значение переменной i
sum         // Возвращает значение переменной sum
undefined   // undefined - глобальная переменная, а не ключевое слово, как null
```

Когда в программе встречается идентификатор, интерпретатор JavaScript предполагает, что это имя переменной и пытается отыскать ее значение. Если переменной с таким именем не существует, возвращается значение `undefined`. Однако в строгом режиме, определяемом стандартом ECMAScript 5, попытка получить значение несуществующей переменной оканчивается исключением `ReferenceError`.

4.2. Инициализаторы объектов и массивов

Инициализаторы объектов и массивов – это выражения, значениями которых являются вновь созданные объекты и массивы. Эти выражения-инициализаторы иногда называют «литералами объектов» и «литералами массивов». Однако, в отличие от истинных литералов, они не являются первичными выражениями, потому что включают множество подвыражений, определяющих значения свойств и элементов. Инициализаторы массивов имеют более простой синтаксис, поэтому мы рассмотрим их в первую очередь.

Инициализатор массива – это список выражений, разделенных запятыми, заключенный в квадратные скобки. Значением инициализатора массива является вновь созданный массив. Элементы этого нового массива инициализируются значениями выражений из списка:

```
[]        // Пустой массив: отсутствие выражений в квадратных скобках
          // означает отсутствие элементов
[1+2,3+4] // Массив из 2 элементов. Первый элемент - 3, второй - 7
```

Выражения в инициализаторе массива, определяющие значения его элементов, сами могут быть инициализаторами массивов, благодаря чему имеется возможность создавать вложенные массивы:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Выражения в инициализаторе массива, определяющие значения его элементов, вычисляются всякий раз, когда вычисляется значение инициализатора. Это означает, что значение выражения инициализатора массива может отличаться при каждом последующем его вычислении.

В литерал массива допускается включать неопределенные элементы, для чего достаточно опустить значение между запятыми. Например, следующий массив содержит пять элементов, включая три элемента с неопределенными значениями:

```
var sparseArray = [1,,,,5];
```

После последнего выражения в инициализаторах массивов допускается указывать завершающую запятую, при этом последний элемент с неопределенным значением создаваться не будет.

Выражения-инициализаторы объектов похожи на выражения-инициализаторы массивов, но вместо квадратных скобок в них используются фигурные скобки, а каждое подвыражение предваряется именем свойства и двоеточием:

```
var p = { x:2.3, y:-1.2 }; // Объект с 2 свойствами
var q = {};                // Пустой объект без свойств
q.x = 2.3; q.y = -1.2;     // Теперь q имеет те же свойства, что и p
```

Литералы объектов могут быть вложенными. Например:

```
var rectangle = { upperLeft: { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 5 } };
```

Выражения внутри инициализаторов объектов вычисляются всякий раз, когда вычисляется значение самого инициализатора, и не обязательно должны быть константами: они могут быть произвольными выражениями JavaScript. Кроме того, имена свойств в литералах объектов могут быть строками, а не идентификаторами (это удобно, когда возникает желание дать свойствам имена, совпадающие с зарезервированными словами, которые иначе не могут использоваться в качестве идентификаторов):

```
var side = 1;
var square = { "upperLeft": { x: p.x, y: p.y },
               'lowerRight': { x: p.x + side, y: p.y + side}};
```

Мы еще вернемся к инициализаторам объектов и массивов в главах 6 и 7.

4.3. Выражения определений функций

Выражение определения функции определяет функцию, а значением такого выражения является вновь созданная функция. В некотором смысле выражение определения функции является «литералом функции» подобно тому, как инициализаторы объектов являются «литералами объектов». Выражение определения функции обычно состоит из ключевого слова `function`, за которым следует список из нуля или более идентификаторов (имен параметров), разделенных запятыми, в круглых скобках и блок программного кода на языке JavaScript (тело функции) в фигурных скобках. Например:

```
// Эта функция возвращает квадрат переданного ей значения
var square = function(x) { return x * x; }
```

Выражение определения функции также может включать имя функции. Кроме того, функции можно определять с помощью инструкции `function`, вместо выражения определения функции. Подробное описание особенностей определения функций приводится в главе 8.

4.4. Выражения обращения к свойствам

Выражение обращения к свойству вычисляет значение свойства объекта или элемента массива. В языке JavaScript имеется два способа обращения к свойствам:

```
выражение . идентификатор
выражение [ выражение ]
```

Первый способ обращения к свойствам заключается в использовании выражения, за которым следуют точка и идентификатор. Выражение определяет объект, а идентификатор – имя требуемого свойства. Первый способ заключается в использовании выражения (объект или массив), за которым следует другое выражение, заключенное в квадратные скобки. Второе выражение определяет имя требуемого свойства или индекс элемента массива. Ниже приводится несколько конкретных примеров:

```
var o = {x:1,y:{z:3}}; // Пример объекта
var a = [o,4,[5,6]];   // Пример массива, содержащего объект
o.x                    // => 1: свойство x выражения o
o.y.z                  // => 3: свойство z выражения o.y
o["x"]                 // => 1: свойство x объекта o
a[1]                   // => 4: элемент с индексом 1 выражения a
a[2]["1"]              // => 6: элемент с индексом 1 выражения a[2]
a[0].x                 // => 1: свойство x выражения a[0]
```

Независимо от способа обращения к свойству первым вычисляется выражение, стоящее перед `.` или `[`. Если значением этого выражения является `null` или `undefined`, возбуждается исключение `TypeError`, потому что эти два значения в JavaScript не имеют свойств. Если значение выражения не является объектом (или массивом), оно будет преобразовано в объект (раздел 3.6). Если за первым выражением следует точка и идентификатор, интерпретатор попытается отыскать значение свойства с именем, совпадающим с идентификатором, которое и станет значением всего выражения. Если за первым выражением следует другое выражение в квадратных скобках, интерпретатор вычислит второе выражение и преобразует его в строку. В этом случае значением всего выражения станет значение свойства, имя которого совпадает со строкой. В любом случае, если свойство с указанным именем не существует, значением выражения обращения к свойству станет значение `undefined`.

Из двух способов обращения к свойству синтаксис `.идентификатор` выглядит проще, но обратите внимание, что этот способ может использоваться, только если именем свойства, к которому выполняется обращение, является допустимый идентификатор, и это имя известно на этапе создания программы. Если имя свойства совпадает с зарезервированным словом, включает пробелы или знаки пунктуации, или когда оно является числом (в случае массивов), необходимо использовать синтаксис с квадратными скобками. Кроме того, квадратные скобки можно использовать, когда имя свойства является не статическим, а результатом некоторых вычислений (пример можно найти в разделе 6.2.1).

Подробнее об объектах и их свойствах рассказывается в главе 6, а массивы и их элементы обсуждаются в главе 7.

4.5. Выражения вызова

Выражение вызова в языке JavaScript служит для вызова (или выполнения) функции или метода. Оно начинается с выражения, возвращающего функцию, идентифицирующего вызываемую функцию. Вслед за выражением получения функции следуют открывающая круглая скобка, список из нуля или более выражений аргументов, разделенных запятыми, и закрывающая круглая скобка. Например:

```
f(0)            // f - выражение функции; 0 - выражение аргумента.
Math.max(x,y,z) // Math.max - функция; x, y и z - аргументы.
a.sort()        // a.sort - функция; здесь нет аргументов.
```

При вычислении выражения вызова первым вычисляется выражение, возвращающее функцию, а затем вычисляются выражения аргументов и создается список значений аргументов. Если значением выражения, возвращающего функцию, не является вызываемый объект, возбуждается исключение `TypeError`. (Все функции являются вызываемыми объектами. Объекты среды выполнения также могут быть вызываемыми, даже если они не являются функциями. Это отличие рассматривается в разделе 8.7.7.) Далее значения аргументов присваиваются в порядке их следования именам параметров, которые указаны в определении функции, после чего выполняется тело функции. Если внутри функции используется инструкция `return`, возвращающая некоторое значение, это значение становится значением выражения вызова. В противном случае выражение вызова возвращает значение `undefined`. Полное описание механизма вызова функций, включая описание того, что происходит, когда количество выражений аргументов не совпадает с количеством параметров в определении функции, вы найдете в главе 8.

Все выражения вызова включают пару круглых скобок и выражение перед открывающей круглой скобкой. Если это выражение является выражением обращения к свойству, такой вызов называется *вызовом метода*. При вызове метода объект или массив, к свойству которого производится обращение, становится значением параметра `this`, доступного в теле функции во время его выполнения. Это обеспечивает поддержку парадигмы объектно-ориентированного программирования, согласно которой функции (в ООП обычно называются «методами») получают возможность манипулировать объектом, частью которого они являются. Подробности приводятся в главе 9.

В выражениях вызова, которые не являются вызовами методов, значением ключевого слова `this` обычно является глобальный объект. Однако согласно стандарту

ECMAScript 5, если функция определяется в строгом режиме, при вызове она получает в ключевом слове `this` не глобальный объект, а значение `undefined`. Подробнее о строгом режиме рассказывается в разделе 5.7.3.

4.6. Выражения создания объектов

Выражение *создания объекта* создает новый объект и вызывает функцию (называемую конструктором) для инициализации свойств этого объекта. Выражения создания объектов похожи на выражения вызова, за исключением того, что им предшествует ключевое слово `new`:

```
new Object()
new Point(2,3)
```

Если в выражении создания объекта функции-конструктору не передается ни одного аргумента, пустую пару круглых скобок можно опустить:

```
new Object
new Date
```

При вычислении выражения создания объекта интерпретатор JavaScript сначала создает новый пустой объект, как если бы для создания использовался пустой инициализатор объекта {}, а затем вызывает указанную функцию с указанными аргументами, передавая ей новый объект в качестве значения ключевого слова `this`. Функция может использовать его для инициализации свойств только что созданного объекта. Функции, которые создаются специально, чтобы играть роль конструктора, не должны возвращать значение, а значением выражения создания объекта становится созданный и инициализированный объект. Если конструктор возвращает какой-либо объект, этот объект становится значением всего выражения создания объекта, а вновь созданный объект уничтожается.

Более подробно конструкторы описываются в главе 9.

4.7. Обзор операторов

В языке JavaScript операторы используются в арифметических выражениях, выражениях сравнения, логических выражениях, выражениях присваивания и т. д. Перечень операторов приводится в табл. 4.1, которую можно использовать как справочник.

Таблица 4.1. Операторы JavaScript

Оператор	Операция	A	N	Типы значений
++	Префиксный и постфиксный инкремент	R	1	левостороннее выражение → число
--	Префиксный и постфиксный декремент	R	1	левостороннее выражение → число
-	Унарный минус	R	1	число → число
+	Преобразование в число	R	1	число → число
~	Поразрядная инверсия	R	1	целое → целое

Оператор	Операция	A	N	Типы значений
`!`	Логическая инверсия	R	1	логическое → логическое
`delete`	Удаление свойства	R	1	левостороннее выражение → логическое
`typeof`	Определение типа операнда	R	1	любое → строка
`void`	Возврат неопределенного значения	R	1	любое → `undefined`
`*, /, %`	Умножение, деление, деление по модулю	L	2	число, число → число
`+, -`	Сложение, вычитание	L	2	число, число → число
`+`	Конкатенация строк	L	2	строка, строка → строка
`<<`	Сдвиг влево	L	2	целое, целое → целое
`>>`	Сдвиг вправо с сохранением знака	L	2	целое, целое → целое
`>>>`	Сдвиг вправо с заполнением нулями	L	2	целое, целое → целое
`<, <=, >, >=`	Сравнение числовых значений	L	2	число, число → логическое
`<, <=, >, >=`	Сравнение строк	L	2	строка, строка → логическое
`instanceof`	Проверка на принадлежность классу	L	2	объект, функция → логическое
`in`	Проверка наличия свойства	L	2	строка, объект → логическое
`==`	Проверка равенства	L	2	любое, любое → логическое
`!=`	Проверка неравенства	L	2	любое, любое → логическое
`===`	Проверка идентичности	L	2	любое, любое → логическое
`!==`	Проверка неидентичности	L	2	любое, любое → логическое
`&`	Поразрядное И	L	2	целое, целое → целое
`^`	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	L	2	целое, целое → целое
`|`	Поразрядное ИЛИ	L	2	целое, целое → целое
`&&`	Логическое И	L	2	любое, любое → любое
`||`	Логическое ИЛИ	L	2	любое, любое → любое
`?:`	Выбор второго или третьего операнда	R	3	логическое, любое, любое → любое
`=`	Присваивание переменной или свойству	R	2	левостороннее выражение, любое → любое
`*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=`	Операция с присваиванием	R	2	левостороннее выражение, любое → любое
`,`	Отбросить первый операнд, вернуть второй	L	2	любое, любое → любое

Обратите внимание, что большинство операторов обозначаются символами пунктуации, такими как `+` и `=`, а некоторые – ключевыми словами, например `delete` и `instanceof`. И ключевые слова, и знаки пунктуации обозначают обычные операторы, просто первые имеют менее лаконичный синтаксис.

Операторы в табл. 4.1 перечислены в порядке их приоритетов. Операторы, перечисленные первыми, имеют более высокий приоритет. Операторы, отделенные горизонтальной линией, имеют разные приоритеты. Столбец «A» в этой таблице содержит ассоциативность оператора (либо L – слева направо, либо R – справа налево), а столбец «N» определяет количество операндов. В столбце «Типы значений» указаны ожидаемые типы операндов и (после символа →) тип результата, возвращаемого оператором. В подразделах, следующих за таблицей, описываются концепции приоритетов, ассоциативности и типов операндов. Вслед за этим приводится обсуждение самих операторов.

4.7.1. Количество операндов

Операторы могут быть разбиты на категории по количеству требуемых им операндов. Большинство JavaScript-операторов, таких как оператор умножения `*`, являются *двухместными*. Такие операторы объединяют два выражения в одно, более сложное. То есть эти операторы работают с двумя операндами. JavaScript поддерживает также несколько *унарных операторов*, которые преобразуют одно выражение в другое, более сложное. Оператор `-` в выражении `-x` является унарным оператором, выполняющим смену знака операнда `x`. И, наконец, JavaScript поддерживает один тернарный условный оператор `?:`, который объединяет три выражения в одно.

4.7.2. Типы данных операндов и результата

Некоторые операторы могут работать со значениями любых типов, но большинство из них требуют, чтобы операнды имели значения определенного типа, и большинство операторов возвращают значение определенного типа. Колонка «Типы значений» в табл. 4.1 определяет типы операндов (перед стрелкой) и тип результата (после стрелки) для операторов.

Операторы в языке JavaScript обычно преобразуют типы своих операндов (как описывается в разделе 3.8) по мере необходимости. Оператор умножения `*` ожидает получить числовые операнды, однако выражение `"3" * "5"` считается вполне допустимым благодаря тому, что интерпретатор выполнит преобразование строковых операндов в числа. Значением этого выражения будет число 15, а не строка `"15"`. Не забывайте также, что любое значение в JavaScript может быть «истинным» или «ложным», поэтому операторы, ожидающие получить логические операнды, будут работать с операндами любого типа.

Некоторые операторы ведут себя по-разному в зависимости от типа операндов. Самый яркий пример – оператор `+`, который складывает числовые операнды и выполняет конкатенацию строк. Аналогично операторы сравнения, такие как `<`, сравнивают значения как числа или как строки, в зависимости от типов операндов. О зависимостях от типов операндов и о выполняемых преобразованиях будет рассказываться в описаниях отдельных операторов.

4.7.3. Левосторонние выражения

Обратите внимание, что операторы присваивания, как и некоторые другие, перечисленные в табл. 4.1, ожидают получить в качестве операндов *левосторонние выражения* (*lvalue*). Левостороннее выражение – это исторический термин, обозначающий «выражение, которое может присутствовать слева от оператора присваивания». В JavaScript левосторонними выражениями являются переменные, свойства объектов и элементы массивов. Спецификация ECMAScript разрешает встроенным функциям возвращать левосторонние выражения, но не определяет никаких встроенных функций, ведущих себя подобным образом.

4.7.4. Побочные эффекты операторов

Вычисление простого выражения, такого как `2 * 3`, никак не отразится на состоянии программы и никак не затронет последующие вычисления, выполняемые программой. Однако некоторые выражения могут иметь *побочные эффекты*, и их вычисление может оказывать влияние на результаты последующих вычислений. Наиболее очевидным примером являются операторы присваивания: если переменной или свойству присвоить некоторое значение, это повлияет на результат любого выражения, в котором используется эта переменная или свойство. Аналогичный побочный эффект имеют операторы инкремента `++` и декремента `--`, поскольку они неявно выполняют присваивание. Оператор `delete` также имеет побочный эффект: операция удаления свойства напоминает (хотя и недостаточно близко) присваивание свойству значения `undefined`.

Никакие другие операторы в языке JavaScript не имеют побочных эффектов, но выражения вызова функции и создания объекта обязательно будут иметь побочные эффекты, если в теле функции или конструктора будут использованы операторы, имеющие побочные эффекты.

4.7.5. Приоритет операторов

Операторы перечислены в табл. 4.1 в порядке уменьшения приоритета, и горизонтальные линии отделяют группы операторов с разным уровнем приоритета. Приоритет оператора управляет порядком, в котором выполняются операции. Операторы с более высоким приоритетом (ближе к началу таблицы) выполняются раньше операторов с более низким приоритетом (ближе к концу таблицы).

Рассмотрим следующее выражение:

```
w = x + y * z;
```

Оператор умножения `*` имеет более высокий приоритет по сравнению с оператором сложения `+`, поэтому умножение выполняется раньше сложения. Оператор присваивания `=` имеет наименьший приоритет, поэтому присваивание выполняется после завершения всех операций в правой части.

Приоритет операторов может быть переопределен с помощью скобок. Чтобы сложение в предыдущем примере выполнялось раньше, надо написать:

```
w = (x + y)* z;
```

Следует учитывать, что выражения вызова и обращения к свойству имеют более высокий приоритет, чем любой из операторов, перечисленных в табл. 4.1. Взгляните на следующее выражение:

```
typeof my.functions[x](y)
```

Несмотря на то что `typeof` является одним из самых высокоприоритетных операторов, операция `typeof` будет выполняться над результатом операций обращения к свойству и вызова функции.

На практике, если вы не уверены в приоритетах применения операторов, проще всего явно задать порядок вычислений с помощью скобок. Важно четко знать следующие правила: умножение и деление выполняются раньше сложения и вычитания, а присваивание имеет очень низкий приоритет и почти всегда выполняется последним.

4.7.6. Ассоциативность операторов

В табл. 4.1 в столбце «А» указана *ассоциативность* операторов. Значение L указывает на ассоциативность слева направо, а значение R – на ассоциативность справа налево. Ассоциативность оператора определяет порядок выполнения операций с одинаковым приоритетом. Ассоциативность слева направо означает, что операции выполняются слева направо. Например, оператор вычитания имеет ассоциативность слева направо, поэтому следующие два выражения эквивалентны:

```
w = x - y - z;
w = ((x - y) - z);
```

С другой стороны, выражения

```
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

эквивалентны следующим выражениям:

```
x = ~(-y);
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

Причина в том, что унарные операторы, операторы присваивания и условные тернарные операторы имеют ассоциативность справа налево.

4.7.7. Порядок вычисления

Приоритет и ассоциативность операторов определяют порядок их выполнения в комплексных выражениях, но они не оказывают влияния на порядок вычислений в подвыражениях. Выражения в языке JavaScript всегда вычисляются слева направо. Например, в выражении `w=x+y*z` первым будет вычислено подвыражение `w`, затем `x`, `y` и `z`. После этого будет выполнено умножение значений `y` и `z`, затем сложение со значением `x` и результат будет присвоен переменной или свойству, определяемому выражением `w`. Добавляя в выражения круглые скобки, можно изменить относительный порядок выполнения операций умножения, сложения и присваивания, но нельзя изменить общий порядок вычислений слева направо.

Порядок вычисления имеет значение, только когда выражение имеет побочные эффекты, оказывающие влияние на значения других выражений. Если выражение `x` увеличивает значение переменной, используемой в выражении `z`, тогда тот факт, что `x` вычисляется раньше, чем `z`, имеет большое значение.

4.8. Арифметические выражения

В этом разделе описываются операторы, выполняющие арифметические и другие операции с числами. Наиболее простыми из них являются операторы умножения, деления и вычитания, поэтому они будут рассмотрены первыми. Оператор сложения будет описан в собственном подразделе, потому что он также выполняет операцию конкатенации строк и использует некоторые необычные правила преобразования типов. Унарные и поразрядные операторы также будут рассматриваться в отдельных подразделах.

Основными арифметическими операторами являются * (умножение), / (деление), % (деление по модулю: остаток от деления), + (сложение) и - (вычитание). Как уже отмечалось, оператор + будет рассматриваться в отдельном разделе. Другие основные четыре оператора просто определяют значения своих операндов, преобразуют их значения в числа, если это необходимо, и вычисляют произведение, частное, остаток или разность значений. Нечисловые операнды, которые не могут быть преобразованы в числа, преобразуются в значение `NaN`. Если какой-либо из операндов имеет (или преобразуется в) значение `NaN`, результатом операции также будет значение `NaN`.

Оператор / делит первый операнд на второй. Если вам приходилось работать с языками программирования, в которых целые и вещественные числа относятся к разным типам, вы могли бы ожидать получить целый результат от деления одного целого числа на другое целое число. Однако в языке JavaScript все числа являются вещественными, поэтому все операции деления возвращают вещественный результат: выражение `5/2` вернет `2.5`, а не `2`. Деление на ноль возвращает положительную или отрицательную бесконечность, тогда как выражение `0/0` возвращает `NaN`; ни в одном из этих случаев не возбуждается исключение.

Оператор `%` производит деление по модулю первого операнда на второй. Иными словами, он возвращает остаток от целочисленного деления первого операнда на второй. Знак результата определяется знаком первого операнда. Например, выражение `5 % 2` вернет `1`, а выражение `-5 % 2` вернет `-1`.

Несмотря на то что оператор по модулю обычно применяется к целым числам, он также может оперировать вещественными значениями. Например, выражение `6.5 % 2.1` вернет 0,2.

4.8.1. Оператор +

Двухместный оператор + складывает числовые операнды или выполняет конкатенацию строковых операндов:

```
1 + 2                     // => 3
"hello" + " " + "there"   // => "hello there"
"1" + "2"                 // => "12"
```

Когда значениями обоих операндов являются числа или строки, результат действия оператора + очевиден. Однако в других случаях возникает необходимость преобразования типов, а выполняемая операция зависит от результатов преобразований. В соответствии с правилами преобразований оператор + отдает предпочтение операции конкатенации строк: если один из операндов является строкой или объектом, который может быть преобразован в строку, другой операнд также преобразуется в строку, после чего выполняется операция конкатенации строк. Операция сложения выполняется, только если ни один из операндов не является строкой.

Формально оператор + использует следующий алгоритм работы:

- Если значением любого из операндов является объект, он преобразуется в простое значение с использованием алгоритма преобразования объекта в простое значение, описанного в разделе 3.8.3: объекты `Date` преобразуются с помощью их метода `toString()`, а все остальные объекты преобразуются с помощью метода `valueOf()`, если он возвращает простое значение. Однако большинство объектов не имеют метода `valueOf()`, поэтому они также преобразуются с помощью метода `toString()`.
- Если после преобразования объекта в простое значение любой из операндов оказывается строкой, другой операнд также преобразуется в строку и выполняется операция конкатенации.
- В противном случае оба операнда преобразуются в числа (или в `NaN`) и выполняется операция сложения.

Например:

```
1 + 2          // => 3: сложение
"1" + "2"      // => "12": конкатенация
"1" + 2        // => "12": конкатенация после преобразования числа в строку
1 + {}         // => "1[object Object]": конкатенация после
               // преобразования объекта в строку
true + true    // => 2: сложение после преобразования логического значения в число
2 + null       // => 2: сложение после преобразования null в 0
2 + undefined  // => NaN: сложение после преобразования undefined в NaN
```

Наконец, важно отметить, что, когда оператор + применяется к строкам и числам, он может нарушать ассоциативность. То есть результат может зависеть от порядка, в каком выполняются операции. Например:

```
1 + 2 + " blind mice";   // => "3 blind mice"
1 + (2 + " blind mice"); // => "12 blind mice"
```

В первом выражении отсутствуют скобки и оператор + имеет ассоциативность слева направо, благодаря чему сначала выполняется сложение двух чисел, а их сумма объединяется со строкой. Во втором выражении порядок выполнения операций изменен с помощью скобок: число 2 объединяется со строкой, давая в результате новую строку. А затем число 1 объединяется с новой строкой, что дает окончательный результат.

4.8.2. Унарные арифметические операторы

Унарные операторы изменяют значение единственного операнда и создают новое значение. Все унарные операторы в JavaScript имеют наивысший приоритет, и все они являются правоассоциативными. Все унарные арифметические операторы, описываемые в этом разделе (+, -, ++ и --), при необходимости преобразуют свой единственный операнд в число. Обратите внимание, что знаки пунктуации + и - используются как унарные и как двухместные операторы.

Ниже перечислены унарные арифметические операторы:

Унарный плюс (+)

Оператор унарного плюса преобразует свой операнд в число (или в `NaN`) и возвращает преобразованное значение. При использовании с числовым операндом он не выполняет никаких действий.

Унарный минус (-)

Когда - используется как унарный оператор, он преобразует свой операнд в число, если это необходимо, и затем изменяет знак результата.

Инкремент (++)

Оператор ++ инкрементирует (т. е. увеличивает на единицу) свой единственный операнд, который должен быть левосторонним выражением (переменной, элементом массива или свойством объекта). Оператор преобразует свой операнд в число, добавляет к этому числу 1 и присваивает результат сложения обратно переменной, элементу массива или свойству.

Значение, возвращаемое оператором ++, зависит от его положения по отношению к операнду. Если поставить его перед операндом (префиксный оператор инкремента), то к операнду прибавляется 1, а результатом является увеличенное значение операнда. Если же он размещается после операнда (постфиксный оператор инкремента), то к операнду прибавляется 1, однако результатом является первоначальное, *неувеличенное* значение операнда. Взгляните на различия в следующих двух выражениях:

```
var i = 1, j = ++i; // i и j содержат значение 2
var i = 1, j = i++; // i содержит значение 2, j содержит значение 1
```

Обратите внимание, что выражение `++x` не всегда возвращает тот же результат, что и выражение `x=x+1`. Оператор ++ никогда не выполняет операцию конкатенации строк: он всегда преобразует свой операнд в число и увеличивает его. Если `x` является строкой «1», выражение `++x` вернет число `2`, тогда как выражение `x+1` вернет строку «11».

Отметьте также, что из-за автоматического добавления точки с запятой в языке JavaScript нельзя вставлять перевод строки между постфиксным оператором инкремента и его операндом. Если это сделать, интерпретатор JavaScript будет рассматривать операнд как полноценную инструкцию и вставит после него точку с запятой.

Данный оператор в обеих своих формах (префиксной и постфиксной) чаще всего применяется для увеличения счетчика, управляющего циклом `for` (раздел 5.5.3).

Декремент (--)

Оператор -- ожидает получить в качестве операнда левостороннее выражение. Он преобразует значение операнда в число, вычитает 1 и присваивает уменьшенное значение обратно операнду. Подобно оператору ++, точное поведение оператора -- зависит от его положения относительно операнда. Если он стоит перед операндом, то уменьшает операнд и возвращает уменьшенное значение; если оператор стоит после операнда, он уменьшает операнд, но возвращает первоначальное, *неуменьшенное* значение. При использовании постфиксной формы операнда не допускается вставлять перевод строки между оператором и операндом.

4.8.3. Поразрядные операторы

Поразрядные операторы выполняют низкоуровневые манипуляции с битами в двоичных представлениях чисел. Несмотря на то что они не выполняют арифметические операции в традиционном понимании, тем не менее они относятся к категории арифметических операторов, потому что оперируют числовыми операндами и возвращают числовое значение. Эти операторы редко используются в программировании на языке JavaScript, и если вы не знакомы с двоичным представлением целых чисел, то можете пропустить этот раздел. Четыре из этих операторов выполняют поразрядные операции булевой алгебры над отдельными битами операндов и действуют так, как если бы каждый бит каждого операнда был представлен логическим значением (1= `true`, 0=`false`). Три других поразрядных оператора применяются для сдвига битов влево и вправо.

Поразрядные операторы работают с целочисленными операндами и действуют так, как если бы эти значения были представлены 32-битными целыми, а не 64-битными вещественными значениями. При необходимости эти операторы преобразуют свои операнды в числа и затем приводят числовые значения к 32-битным целым, отбрасывая дробные части и любые биты старше 32-го. Операторы сдвига требуют, чтобы значение правого операнда было целым числом от 0 до 31. После преобразования этого операнда в 32-битное беззнаковое целое они отбрасывают любые биты старше 5-го, получая число в соответствующем диапазоне. Самое интересное, что эти операторы преобразуют значения `NaN`, `Infinity` и `-Infinity` в 0.

Поразрядное И (`&`)

> Оператор `&` выполняет операцию «логическое И» над каждым битом своих целочисленных аргументов. Бит результата устанавливается, если соответствующий бит установлен в обоих операндах. Например, выражение `0x1234 & 0x00FF` даст в результате число `0x0034`.

Поразрядное ИЛИ (`|`)

> Оператор `|` выполняет операцию «логическое ИЛИ» над каждым битом своих целочисленных аргументов. Бит результата устанавливается, если соответствующий бит установлен хотя бы в одном операнде. Например, выражение `0x1234 | 0x00FF` даст в результате `0x12FF`.

Поразрядное исключающее ИЛИ (`^`)

> Оператор `^` выполняет логическую операцию «исключающее ИЛИ» над каждым битом своих целочисленных аргументов. Исключающее ИЛИ означает, что должен быть истинен либо первый операнд, либо второй, но не оба сразу. Бит результата устанавливается, если соответствующий бит установлен в одном (но не в обоих) из двух операндов. Например, выражение `0xFF00 ^ 0xF0F0` даст в результате `0x0FF0`.

Поразрядное НЕ (`~`)

> Оператор `~` представляет собой унарный оператор, указываемый перед своим единственным целым операндом. Он выполняет инверсию всех битов операнда. Из-за способа представления целых со знаком в JavaScript применение оператора `~` к значению эквивалентно изменению его знака и вычитанию 1. Например, выражение `~0x0f` даст в результате `0xFFFFFFF0`, или -16.

Сдвиг влево (`<<`)

> Оператор `<<` сдвигает все биты в первом операнде влево на количество позиций, указанное во втором операнде, который должен быть целым числом в диапазоне от 0 до 31. Например, в операции `a << 1` первый бит в `a` становится вторым битом, второй бит становится третьим и т. д. Новым первым битом становит-

ся ноль, значение 32-го бита теряется. Сдвиг значения влево на одну позицию эквивалентен умножению на 2, на две позиции – умножению на 4 и т. д. Например, выражение `7 << 2` даст в результате 28.

Сдвиг вправо с сохранением знака (>>)

Оператор >> сдвигает все биты своего первого операнда вправо на количество позиций, указанное во втором операнде (целое между 0 и 31). Биты, сдвинутые за правый край, теряются. Самый старший бит не изменяется, чтобы сохранить знак результата. Если первый операнд положителен, старшие биты результата заполняются нулями; если первый операнд отрицателен, старшие биты результата заполняются единицами. Сдвиг значения вправо на одну позицию эквивалентен делению на 2 (с отбрасыванием остатка), сдвиг вправо на две позиции эквивалентен делению на 4 и т. д. Например, выражение `7 >> 1` даст в результате 3, а выражение `-7 >> 1` даст в результате -4.

Сдвиг вправо с заполнением нулями (>>>)

Оператор >>> аналогичен оператору >>, за исключением того, что при сдвиге старшие разряды заполняются нулями, независимо от знака первого операнда. Например, выражение `-1 >> 4` даст в результате -1, а выражение `-1 >>> 4` даст в результате `0X0FFFFFFF`.

4.9. Выражения отношений

В этом разделе описаны операторы отношения в языке JavaScript. Это операторы проверяют отношение между двумя значениями (такое как «равно», «меньше» или «является ли свойством») и возвращают `true` или `false` в зависимости от того, как соотносятся операнды. Выражения отношений всегда возвращают логические значения, и эти значения чаще всего применяются в инструкциях `if`, `while` и `for` для управления ходом исполнения программы (глава 5). В следующих подразделах описываются операторы равенства и неравенства, операторы сравнения и два других оператора отношений, `in` и `instanceof`.

4.9.1. Операторы равенства и неравенства

Операторы == и === проверяют два значения на совпадение, используя два разных определения совпадения. Оба оператора принимают операнды любого типа и возвращают `true`, если их операнды совпадают, и `false`, если они различны. Оператор ===, известный как оператор идентичности, проверяет два операнда на «идентичность», руководствуясь строгим определением совпадения. Оператор ==, оператор равенства, проверяет, равны ли два его операнда в соответствии с менее строгим определением совпадения, допускающим преобразования типов.

В языке JavaScript поддерживаются операторы =, == и ===. Убедитесь, что вы понимаете разницу между операторами присваивания, равенства и идентичности. Будьте внимательны и применяйте правильные операторы при разработке своих программ! Очень заманчиво назвать все три оператора «равно», но во избежание путаницы лучше читать оператор = как «получается», или «присваивается», оператор == читать как «равно», а словом «идентично» обозначать оператор ===.

Операторы != и !== выполняют проверки, в точности противоположные операторам == и ===. Оператор неравенства != возвращает `false`, если два значения равны друг другу в том смысле, в каком они считаются равными оператором `==`, и `true` в противном случае. Как будет рассказываться в разделе 4.10, оператор `!` выполняет логическую операцию НЕ. Отсюда легко будет запомнить, что операторы `!=` и `!==` означают «не равно» и «не идентично».

Как отмечалось в разделе 3.7, объекты в языке JavaScript сравниваются по ссылке, а не по значению. Это значит, что объект равен только сам себе и не равен никакому другому объекту. Даже если два различных объекта обладают одним и тем же набором свойств, с теми же именами и значениями, они все равно будут считаться неравными. Два массива никогда не могут быть равными, даже если они содержат одинаковые элементы, следующие в одном порядке.

Оператор идентичности `===` вычисляет значения своих операндов, а затем сравнивает два значения, без преобразования типов, руководствуется следующими правилами:

- Если два значения имеют различные типы, они не идентичны.
- Если оба операнда являются значением `null` или `undefined`, они идентичны.
- Если оба операнда являются логическим значением `true` или оба являются логическим значением `false`, они идентичны.
- Если одно или оба значения являются значением `NaN`, они не идентичны. Значение `NaN` никогда не бывает идентичным никакому значению, даже самому себе! Чтобы проверить, является ли значение `x` значением `NaN`, следует использовать выражение `x !== x`. Значение `NaN` – единственное, для которого такая проверка вернет `true`.
- Если оба значения являются числами с одним и тем же значением, они идентичны. Если один операнд имеет значение `0`, а другой `-0`, они также идентичны.
- Если оба значения являются строками и содержат одни и те же 16-битные значения (подробности во врезке в разделе 3.2) в одинаковых позициях, они идентичны. Если строки отличаются длиной или содержимым, они не идентичны. Две строки могут иметь один и тот же смысл и одинаково выглядеть на экране, но содержать отличающиеся последовательности 16-битных значений. Интерпретатор JavaScript не выполняет нормализацию символов Юникода, поэтому подобные пары строк не считаются операторами `===` и `==` ни равными, ни идентичными. Другой способ сравнения строк обсуждается в части III книги, в описании метода `String.localeCompare()`.
- Если оба значения ссылаются на один и тот же объект, массив или функцию, то они идентичны. Если они ссылаются на различные объекты (массивы или функции), они не идентичны, даже если оба объекта имеют идентичные свойства.

Оператор равенства `==` похож на оператор идентичности, но он использует менее строгие правила. Если значения операндов имеют разные типы, он выполняет преобразование типов и пытается выполнить сравнение:

- Если два значения имеют одинаковый тип, они проверяются на идентичность, как было описано выше. Если значения идентичны, они равны; если они не идентичны, они не равны.
- Если два значения не относятся к одному и тому же типу, оператор `==` все же может счесть их равными. При этом используются следующие правила и преобразования типов:

- Если одно значение `null`, а другое – `undefined`, то они равны.
- Если одно значение является числом, а другое – строкой, то строка преобразуется в число и выполняется сравнение с преобразованным значением.
- Если какое-либо значение равно `true`, оно преобразуется в 1 и сравнение выполняется снова. Если какое-либо значение равно `false`, оно преобразуется в 0 и сравнение выполняется снова.
- Если одно из значений является объектом, а другое – числом или строкой, объект преобразуется в простой тип (как описывалось в разделе 3.8.3) и сравнение выполняется снова. Объект преобразуется в значение простого типа либо с помощью своего метода `toString()`, либо с помощью своего метода `valueOf()`. Встроенные классы базового языка JavaScript сначала пытаются выполнить преобразование `valueOf()`, а затем `toString()`, кроме класса `Date`, который всегда выполняет преобразование `toString()`. Объекты, не являющиеся частью базового JavaScript, могут преобразовывать себя в значения простых типов способом, определенным их реализацией.
- Любые другие комбинации значений не являются равными.

В качестве примера проверки на равенство рассмотрим сравнение:

```
"1" == true
```

Результат этого выражения равен `true`, т. е. эти по-разному выглядящие значения фактически равны. Логическое значение `true` преобразуется в число 1, и сравнение выполняется снова. Затем строка `"1"` преобразуется в число `1`. Поскольку оба числа теперь совпадают, оператор сравнения возвращает `true`.

4.9.2. Операторы сравнения

Операторы сравнения определяют относительный порядок двух величин (числовых или строковых):

Меньше (<)

Оператор < возвращает `true`, если первый операнд меньше, чем второй операнд; в противном случае он возвращает `false`.

Больше (>)

Оператор > возвращает `true`, если его первый операнд больше, чем второй операнд; в противном случае он возвращает `false`.

Меньше или равно (<=)

Оператор <= возвращает `true`, если первый операнд меньше или равен второму операнду; в противном случае он возвращает `false`.

Больше или равно (>=)

Оператор >= возвращает `true`, если его первый операнд больше второго или равен ему; в противном случае он возвращает `false`.

Эти операторы позволяют сравнивать операнды любого типа. Однако сравнение может выполняться только для чисел и строк, поэтому операнды, не являющиеся числами или строками, преобразуются. Сравнение и преобразование выполняются следующим образом:

- Если какой-либо операнд является объектом, этот объект преобразуется в простое значение, как было описано в конце раздела 3.8.3: если метод `valueOf()` объекта возвращает простое значение, используется это значение. В противном случае используется значение, возвращаемое методом `toString()`.
- Если после преобразований объектов в простые значения оба операнда оказываются строками, они сравниваются как строки в соответствии с алфавитным порядком, где под «алфавитным порядком» понимается числовой порядок 16-битных значений кодовых пунктов Юникода, составляющих строки.
- Если после преобразований объектов в простые значения хотя бы один операнд не является строкой, оба операнда преобразуются в числа и сравниваются как числа. Значения `0` и `-0` считаются равными. Значение `Infinity` считается больше любого другого числа, а значение `-Infinity` – меньше любого другого числа. Если какой-либо из операндов преобразуется в значение `NaN`, то оператор сравнения всегда возвращает `false`.

Не забывайте, что строки в JavaScript являются последовательностями 16-битных целочисленных значений, и сравнение строк фактически сводится к числовому сравнению этих значений в строках. Порядок кодирования символов, определяемый стандартом Юникода, может не совпадать с традиционным алфавитным порядком, используемым в конкретных языках или регионах. Обратите внимание, что сравнение строк производится с учетом регистра символов и все прописные буквы в кодировке ASCII «меньше» соответствующих им строчных букв ASCII. Это правило может приводить к непонятным результатам. Например, согласно оператору < строка `"Zoo"` меньше строки `"aardvark"`.

При сравнении строк более надежные результаты позволяет получить метод `String.localeCompare()`, который учитывает национальные определения «алфавитного порядка». Для сравнения без учета регистра необходимо сначала преобразовать строки в нижний или верхний регистр с помощью метода `String.toLowerCase()` или `String.toUpperCase()`.

Оператор + и операторы сравнения по-разному обрабатывают числовые и строковые операнды. Оператор + отдает предпочтение строкам: если хотя бы один из операндов является строкой, он выполняет конкатенацию строк. Операторы сравнения отдают предпочтение числам и выполняют строковое сравнение, только когда оба операнда являются строками:

```
1 + 2       // Сложение. Результат: 3.
"1" + "2"   // Конкатенация. Результат: "12".
"1" + 2     // Конкатенация. 2 преобразуется в "2". Результат: "12".
11 < 3      // Числовое сравнение. Результат: false.
"11" < "3"  // Строковое сравнение. Результат: true.
"11" < 3    // Числовое сравнение. "11" преобразуется в 11. Результат: false
"one" < 3   // Числовое сравнение. "one" преобразуется в NaN. Результат: false.
```

Наконец, обратите внимание, что операторы <= (меньше или равно) и >= (больше или равно) для определения «равенства» двух значений не используют операторы равенства или идентичности. Оператор «меньше или равно» определяется просто как «не больше», а оператор «больше или равно» – как «не меньше». Единственное исключение имеет место, когда один из операндов представляет собой значение `NaN` (или преобразуется в него). В этом случае все четыре оператора сравнения возвращают `false`.

4.9.3. Оператор in

Оператор `in` требует, чтобы левый операнд был строкой или мог быть преобразован в строку. Правым операндом должен быть объект. Результатом оператора будет значение `true`, если левое значение представляет собой имя свойства объекта, указанного справа. Например:

```
var point = { x:1, y:1 }; // Определить объект
"x" in point              // => true: объект имеет свойство с именем "x"
"z" in point              // => false: объект не имеет свойства с именем "z".
"toString" in point       // => true: объект наследует метод toString

var data = [7,8,9];       // Массив с элементами 0, 1 и 2
"0" in data               // => true: массив содержит элемент "0"
1 in data                 // => true: числа преобразуются в строки
3 in data                 // => false: нет элемента 3
```

4.9.4. Оператор instanceof

Оператор `instanceof` требует, чтобы левым операндом был объект, а правым – имя класса объектов. Результатом оператора будет значение `true`, если объект, указанный слева, является экземпляром класса, указанного справа. В противном случае результатом будет `false`. В главе 9 рассказывается, что классы объектов в языке JavaScript определяются инициализировавшей их функцией-конструктором. Следовательно, правый операнд оператора `instanceof` должен быть именем функции-конструктора. Например:

```
var d = new Date();       // Создать новый объект с помощью конструктора Date()
d instanceof Date;        // Вернет true; объект d был создан с функцией Date()
d instanceof Object;      // Вернет true; все объекты являются экземплярами Object
d instanceof Number;      // Вернет false; d не является объектом Number
var a = [1, 2, 3];        // Создать массив с помощью литерала массива
a instanceof Array;       // Вернет true; a - это массив
a instanceof Object;      // Вернет true; все массивы являются объектами
a instanceof RegExp;      // Вернет false; массивы не являются регулярными выражениями
```

Обратите внимание, что все объекты являются экземплярами класса `Object`. Определяя, является ли объект экземпляром класса, оператор `instanceof` принимает во внимание и «суперклассы». Если левый операнд `instanceof` не является объектом, `instanceof` возвращает `false`. Если правый операнд не является функцией, возбуждается исключение `TypeError`.

Чтобы понять, как действует оператор `instanceof`, необходимо познакомиться с таким понятием, как «цепочка прототипов». Это – механизм наследования в JavaScript; он описывается в разделе 6.2.2. Чтобы вычислить значение выражения `o instanceof f`, интерпретатор JavaScript определяет значение `f.prototype` и затем пытается отыскать это значение в цепочке прототипов объекта `o`. В случае успеха объект `o` считается экземпляром класса `f` (или суперкласса класса `f`), и оператор возвращает `true`. Если значение `f.prototype` отсутствует в цепочке прототипов объекта `o`, то объект `o` не является экземпляром класса `f` и оператор `instanceof` возвращает `false`.

4.10. Логические выражения

Логические операторы `&&`, `||` и `!` используются для выполнения операций булевой алгебры и часто применяются в сочетании с операторами отношений для объединения двух выражений отношений в одно более сложное выражение. Эти операторы описываются в подразделах, следующих ниже. Чтобы понять, как они действуют, вам может потребоваться еще раз прочитать о концепции «истинности» и «ложности» значений в разделе 3.3.

4.10.1. Логическое И (&&)

Условно говоря, оператор `&&` действует на трех уровнях. На самом простом уровне, когда в операции участвуют логические операнды, оператор `&&` выполняет операцию «логическое И» над двумя значениями: он возвращает `true` тогда и только тогда, когда оба операнда имеют значение `true`. Если один или оба операнда имеют значение `false`, оператор возвращает `false`.

Оператор `&&` часто используется для объединения двух выражений отношений:

```
x == 0 && y == 0   // true тогда и только тогда, когда x и y равны 0
```

Выражения отношений всегда возвращают значение `true` или `false`, поэтому в подобных ситуациях сам оператор `&&` всегда возвращает `true` или `false`. Операторы отношений имеют более высокий приоритет, чем оператор `&&` (и `||`), поэтому такие выражения можно записывать без использования скобок.

Но оператор `&&` не требует, чтобы его операнды были логическими значениями. Напомню, что все значения в языке JavaScript являются либо «истинными», либо «ложными». (Подробности в разделе 3.3. Ложными значениями являются `false`, `null`, `undefined`, `0`, `-0`, `NaN` и `""`. Все другие значения, включая все объекты, являются истинными.) На втором уровне оператор `&&` действует как логическое И для истинных и ложных значений. Если оба операнда являются истинными, оператор возвращает истинное значение. В противном случае, когда один или оба операнда являются ложными, возвращается ложное значение. В языке JavaScript все выражения и инструкции, использующие логические значения, будут также работать с истинными или ложными значениями, поэтому тот факт, что оператор `&&` не всегда возвращает `true` или `false`, на практике не вызывает никаких проблем.

Обратите внимание, что в предыдущем абзаце говорилось, что оператор возвращает «истинное значение» или «ложное значение», но при этом не уточнялось, какое именно значение возвращается. Для этого нам необходимо перейти на третий, заключительный уровень оператора `&&`. Свою работу оператор начинает с вычисления первого операнда – выражения слева. Если выражение слева возвращает ложное значение, значением всего выражения также должно быть ложное значение, поэтому оператор `&&` просто возвращает значение слева и не вычисляет выражение справа.

В противном случае, если значение слева является истинным, тогда результат всего выражения определяется значением справа. Если значение справа является истинным, значением всего выражения также должно быть истинное значение, а если значение справа является ложным, значением всего выражения должно быть ложное значение. Поэтому, когда значение слева является истинным, оператор `&&` вычисляет и возвращает значение справа:

```
var o = { x : 1 };
var p = null;
o && o.x              // => 1: o - истинное значение, поэтому возвращается o.x
p && p.x              // => null: p - ложное значение, поэтому возвращается p,
                      // а выражение p.x не вычисляется
```

Важно понимать, что оператор `&&` может не вычислять выражение правого операнда. В примере выше переменная `p` имеет значение `null`, поэтому попытка вычислить выражение `p.x` привела бы к ошибке `TypeError`. Но здесь задействован оператор `&&`, благодаря чему выражение `p.x` вычисляется, только если `p` будет содержать истинное значение – не `null` или `undefined`.

Такое поведение оператора `&&` иногда называют «короткой схемой вычислений», и иногда можно встретить программный код, в котором такое поведение оператора `&&` используется специально для выполнения инструкций по условию. Например, следующие две строки дают одинаковый результат:

```
if (a == b) stop();   // Функция stop() вызывается, только если a == b
(a == b) && stop();   // То же самое
```

В целом следует с осторожностью использовать выражения с побочными эффектами (присваивания, инкременты, декременты или вызовы функций) справа от оператора `&&`, потому что эти побочные эффекты будут проявляться в зависимости от значения слева.

Несмотря на довольно запутанный алгоритм работы этого оператора, проще всего и абсолютно безопасно рассматривать его как оператор булевой алгебры, который манипулирует истинными и ложными значениями.

4.10.2. Логическое ИЛИ (||)

Оператор || выполняет операцию «логическое ИЛИ» над двумя операндами. Если один или оба операнда имеют истинное значение, он возвращает истинное значение. Если оба операнда имеют ложные значения, он возвращает ложное значение.

Хотя оператор `||` чаще всего применяется просто как оператор «логическое ИЛИ», он, как и оператор `&&`, ведет себя более сложным образом. Его работа начинается с вычисления первого операнда, выражения слева. Если значение этого операнда является истинным, возвращается истинное значение. В противном случае оператор вычисляет второй операнд, выражение справа, и возвращает значение этого выражения.

Как и при использовании оператора `&&`, следует избегать правых операндов, имеющих побочные эффекты, если только вы умышленно не хотите воспользоваться тем обстоятельством, что выражение справа может не вычисляться.

Характерное использование этого оператора заключается в том, что он выбирает первое истинное значение из предложенного множества альтернатив:

```
// Если переменная max_width определена, используется ее значение. В противном случае
// значение извлекается из объекта preferences. Если объект (или его свойство max_with)
// не определен, используется значение константы, жестко определенной в тексте программы.
var max = max_width || preferences.max_width || 500;
```

Этот прием часто используется в функциях для определения значений по умолчанию параметров:

```
// Скопировать свойства объекта o в объект p и вернуть p
function copy(o, p) {
    p = p || {}; // Если объект p не был передан, создать новый объект.
    // реализация тела функции
}
```

4.10.3. Логическое НЕ (!)

Оператор `!` является унарным оператором, помещаемым перед одиночным операндом. Он используется для инверсии логического значения своего операнда. Например, если переменная `x` имеет истинное значение, то выражение `!x` возвращает значение `false`. Если `x` имеет ложное значение, то выражение `!x` возвращает значение `false`.

В отличие от операторов `&&` и `||`, оператор `!` преобразует свой операнд в логическое значение (используя правила, описанные в главе 3) перед тем, как инвертировать его. Это означает, что оператор `!` всегда возвращает `true` или `false` что всегда можно преобразовать любое значение `x` в его логический эквивалент, дважды применив этот оператор: `!!x` (раздел 3.8.2).

Будучи унарным, оператор `!` имеет высокий приоритет и тесно связан с операндом. Если вам потребуется инвертировать значение выражения, такого как `p && q`, необходимо будет использовать круглые скобки: `!(p && q)`. В булевой алгебре есть две теоремы, которые можно выразить на языке JavaScript:

```
// Следующие две проверки на идентичность дают положительный результат
// при любых значениях p и q
!(p && q) === !p || !q
!(p || q) === !p && !q
```

4.11. Выражения присваивания

Для присваивания значения переменной или свойству в языке JavaScript используется оператор `=`. Например:

```
i = 0       // Присвоит переменной i значение 0.
o.x = 1     // Присвоит свойству x объекта o значение 1.
```

Левым операндом оператора `=` должно быть левостороннее выражение: переменная, элемент массива или свойство объекта. Правым операндом может быть любое значение любого типа. Значением оператора присваивания является значение правого операнда. Побочный эффект оператора `=` заключается в присваивании значения правого операнда переменной или свойству, указанному слева, так что при последующих обращениях к переменной или свойству будет получено это значение.

Чаще всего выражения присваивания используются как самостоятельные инструкции; тем не менее иногда можно увидеть, как выражение присваивания включается в более сложные выражения. Например, в одном выражении можно совместить операции присваивания и проверки значения:

```
(a = b) == 0
```

При этом следует отчетливо понимать, что между операторами `=` и `==` есть разница! Обратите внимание, что оператор `=` имеет самый низкий приоритет, поэтому обычно бывает необходимо использовать круглые скобки, если выражение присваивания используется в составе более сложного выражения.

Оператор присваивания имеет ассоциативность справа налево, поэтому при наличии в выражении нескольких операторов присваивания они вычисляются справа налево. Благодаря этому можно написать код, присваивающий одно значение нескольким переменным, например:

```
i = j = k = 0; // Инициализировать 3 переменные значением 0
```

4.11.1. Присваивание с операцией

Помимо обычного оператора присваивания = в языке JavaScript поддерживается несколько других операторов, объединяющих присваивание с некоторой другой операцией. Например, оператор += выполняет сложение и присваивание. Следующее выражение:

```
total += sales_tax
```

эквивалентно выражению:

```
total = total + sales_tax
```

Как можно было ожидать, оператор += работает и с числами, и со строками. Для числовых операндов он выполняет сложение и присваивание, а для строковых — конкатенацию и присваивание.

Из подобных ему операторов можно назвать -=, *=, &= и др. Все операторы присваивания с операцией перечислены в табл. 4.2.

Таблица 4.2. Операторы присваивания

Оператор	Пример	Эквивалент
+=	`a += b`	`a = a + b`
-=	`a -= b`	`a = a - b`
*=	`a *= b`	`a = a * b`
/=	`a /= b`	`a = a / b`
%=	`a %= b`	`a = a % b`
<<=	`a <<= b`	`a = a << b`
>>=	`a >>= b`	`a = a >> b`
>>>=	`a >>>= b`	`a = a >>> b`
&=	`a &= b`	`a = a & b`
|=	`a |= b`	`a = a | b`
^=	`a ^= b`	`a = a ^ b`

В большинстве случаев выражение:

```
a op= b
```

где *op* означает оператор, эквивалентно выражению:

```
a = a op b
```

В первой строке выражение `a` вычисляется только один раз. Во второй – дважды. Эти выражения отличаются, только если подвыражение `a` содержит операции, имеющие побочные эффекты, такие как вызов функции или оператор инкремента. Например, следующие два выражения присваивания неэквивалентны:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12. Вычисление выражений

Подобно многим интерпретирующим языкам, JavaScript поддерживает возможность интерпретации строк с программным кодом на языке JavaScript, выполняя их, чтобы получить некоторое значение. В JavaScript эта операция выполняется с помощью глобальной функции `eval()`:

```
eval("3+2")    // => 5
```

Динамическое выполнение строк с программным кодом является мощной особенностью языка, которая практически никогда не требуется на практике. Если у вас появится желание использовать функцию `eval()`, задумайтесь – действительно ли это необходимо.

В подразделах ниже описываются основы использования функции `eval()` и затем рассказывается о двух ее ограниченных версиях, которые оказывают меньше влияния на оптимизатор.

eval() – функция или оператор?

`eval()` является функцией, но знакомство с ней включено в главу, описывающую операторы, потому что в действительности она должна была бы быть оператором. В самых ранних версиях языка `eval()` определялась как функция, но с тех пор проектировщики языка и разработчики интерпретаторов наложили на нее столько ограничений, что она стала больше похожа на оператор. Современные интерпретаторы JavaScript детально анализируют программный код и выполняют множество оптимизаций. Проблема функции `eval()` заключается в том, что программный код, который она выполняет, в целом не доступен для анализа. Вообще говоря, если функция вызывает `eval()`, интерпретатор не может оптимизировать эту функцию. Проблема с определением `eval()` как функции заключается в том, что ей можно присвоить другие имена:

```
var f = eval;
var g = f;
```

Если допустить такую возможность, интерпретатор не сможет обеспечить безопасность оптимизации любых функций, вызывающих `g()`. Данной проблемы можно было бы избежать, если бы `eval` была оператором (и зарезервированным словом). С ограничениями, накладываемыми на функцию `eval()`, которые делают ее более похожей на оператор, мы познакомимся в разделах ниже (разделы 4.12.2 и 4.12.3).

4.12.1. eval()

Функция `eval()` принимает единственный аргумент. Если передать ей значение, отличное от строки, она просто вернет это значение. Если передать ей строку, она попытается выполнить синтаксический анализ этой строки как программного кода на языке JavaScript и возбудит исключение `SyntaxError` в случае неудачи. В случае успеха она выполнит этот программный код и вернет значение последнего выражения или инструкции в строке либо значение `undefined`, если последнее выражение или инструкция не имеют значения. Если программный код в строке возбудит исключение, функция `eval()` передаст это исключение дальше.

Ключевой особенностью функции `eval()` (когда она вызывается таким способом) является то обстоятельство, что она использует окружение программного кода, вызвавшего ее. То есть она будет отыскивать значения переменных и определять новые переменные и функции, как это делает локальный программный код. Если функция определит локальную переменную `x` и затем вызовет `eval("x")`, она получит значение локальной переменной. Вызов `eval("x=1")` изменит значение локальной переменной. А если выполнить вызов `eval("var y = 3;")`, будет объявлена новая локальная переменная `y`. Точно так же можно определять новые локальные функции:

```
eval("function f() { return x+1; }");
```

Если вызвать функцию `eval()` из программного кода верхнего уровня, она, разумеется, будет оперировать глобальными переменными и глобальными функциями.

Обратите внимание, что программный код в строке, передаваемой функции `eval()`, должен быть синтаксически осмысленным – эту функцию нельзя использовать, чтобы вставить фрагмент программного кода в вызывающую функцию. Например, бессмысленно писать вызов `eval("return;")`, потому что инструкция `return` допустима только внутри функций, а тот факт, что программный код в строке использует то же самое окружение, что и вызывающая функция, не делает его частью этой функции. Если программный код в строке может расцениваться как самостоятельный сценарий (пусть и очень короткий, такой как `x=0`), его уже можно будет передавать функции `eval()`. В противном случае `eval()` возбудит исключение `SyntaxError`.

4.12.2. Использование eval() в глобальном контексте

Способность функции `eval()` изменять локальные переменные представляет значительную проблему для оптимизаторов JavaScript. Для ее решения некоторые интерпретаторы просто уменьшают степень оптимизации всех функций, вызывающих `eval()`. Однако как быть интерпретатору JavaScript, когда в сценарии определяется псевдоним функции `eval()` и выполняется ее вызов по другому имени? Чтобы облегчить жизнь разработчикам интерпретаторов JavaScript, стандарт ECMAScript 3 требует, чтобы такая возможность в интерпретаторах была запрещена. Если функция `eval()` вызывается под любым другим именем, отличным от «eval», она должна возбуждать исключение `EvalError`.

Однако большинство разработчиков используют иные решения. При вызове под любым другим именем функция `eval()` должна выполнять программный код в глобальном контексте. Выполняемый ею программный код может определять новые глобальные переменные или глобальные функции и изменять значения глобальных переменных, но не может использоваться для модификации локальных переменных в вызывающих функциях, благодаря чему устраняются препятствия для локальной оптимизации.

Стандарт ECMAScript 5 отменяет возбуждение исключения `EvalError` и стандартизует поведение `eval()`, сложившееся де-факто. «Прямой вызов» – это вызов функции `eval()` по ее непосредственному имени «eval» (которое все больше начинает походить на зарезервированное слово). Прямые вызовы `eval()` используют окружение вызывающего контекста. Любые другие вызовы – косвенные вызовы – в качестве окружения используют глобальный объект и не могут получать, изменять или определять локальные переменные или функции. Это поведение демонстрируется в следующем фрагменте:

```
var geval = eval;                  // Другое имя eval для вызова в глобальном контексте
var x = "global", y = "global";    // Две глобальные переменные

function f() {                     // Вызывает eval в локальном контексте
    var x = "local";               // Определение локальной переменной
    eval("x += 'changed';");       // Прямой вызов eval изменит локальную переменную
    return x;                      // Вернет измененную локальную переменную
}
function g() {                     // Вызывает eval в глобальном контексте
    var y = "local";               // Локальная переменная
    geval("y += 'changed';");      // Косвенный вызов eval изменит глоб. переменную
    return y;                      // Вернет неизмененную локальную переменную
}
console.log(f(), x); // Изменилась локальная переменная: выведет "localchanged global":
console.log(g(), y); // Изменилась глобальная переменная: выведет "local globalchanged":
```

Обратите внимание, что обеспечение возможности вызывать функцию `eval()` в глобальном контексте – это не просто попытка удовлетворить потребности оптимизатора. Фактически это чрезвычайно полезная особенность: она позволяет выполнять строки с программным кодом, как если бы они были независимыми сценариями. Как уже отмечалось в начале этого раздела, действительная необходимость выполнять строки программного кода на практике возникает очень редко. Но если вы сочтете это необходимым, вам, скорее всего, потребуется вызывать функцию `eval()` именно в глобальном контексте, а не в локальном.

До появления версии IE9 Internet Explorer отличался от других броузеров: функция `eval()`, вызванная под другим именем, выполняла переданный ей программный код не в глобальном контексте. (Однако она не возбуждала исключение `EvalError`: программный код просто выполнялся ею в локальном контексте.) Но IE определяет глобальную функцию `execScript()`, которая выполняет строку с программным кодом, переданную в виде аргумента, как если бы она была сценарием верхнего уровня. (Однако, в отличие от `eval()`, функция `execScript()` всегда возвращает `null`.)

4.12.3. Использование eval() в строгом режиме

Строгий режим (раздел 5.7.3), определяемый стандартом ECMAScript 5, вводит дополнительные ограничения на поведение функции `eval()` и даже на использование идентификатора «eval». Когда функция `eval()` вызывается из программного кода, выполняемого в строгом режиме, или когда строка, которая передается функции, начинается с директивы «use strict», то `eval()` выполняет программный код в частном окружении. Это означает, что в строгом режиме выполняемый программный код может обращаться к локальным переменным и изменять их, но он не может определять новые переменные или функции в локальной области видимости.

Кроме того, строгий режим делает функцию `eval()` еще более похожей на оператор, фактически превращая «eval» в зарезервированное слово. В этом режиме нельзя переопределить функцию `eval()` новым значением. А также нельзя объявить переменную, функцию, параметр функции или параметр блока `catch` с именем «eval».

4.13. Прочие операторы

JavaScript поддерживает еще несколько операторов, которые описываются в следующих разделах.

4.13.1. Условный оператор (?:)

Условный оператор – это единственный тернарный (с тремя операндами) оператор в JavaScript, и иногда он так и называется – «тернарный оператор». Этот оператор обычно записывается как `?:`, хотя в программах он выглядит по-другому. Он имеет три операнда, первый предшествует символу `?`, второй – между `?` и `:`, третий – после `:`. Используется он следующим образом:

```
x > 0 ? x : -x     // Абсолютное значение x
```

Операнды условного оператора могут быть любого типа. Первый операнд вычисляется и используется как логическое значение. Если первый операнд имеет истинное значение, то вычисляется и возвращается значение выражения во втором операнде. Если первый операнд имеет ложное значение, то вычисляется и возвращается значение выражения в третьем операнде. Вычисляется всегда только какой-то один операнд, второй или третий, и никогда оба.

Тот же результат можно получить с помощью инструкции `if`, но оператор `?:` часто оказывается удобным сокращением. Ниже приводится типичный пример, в котором проверяется, определена ли переменная (и имеет истинное значение), и если да, то берется ее значение, а если нет, берется значение по умолчанию:

```
greeting = "hello " + (username ? username : "there");
```

Эта проверка эквивалентна следующей конструкции `if`, но более компактна:

```
greeting = "hello ";
if (username)
   greeting += username;
else
   greeting += "there";
```

4.13.2. Оператор typeof

Унарный оператор `typeof` помещается перед единственным операндом, который может иметь любой тип. Его значением является строка, указывающая на тип данных операнда. Следующая таблица определяет значения оператора `typeof` для всех значений, возможных в языке JavaScript:

x	**typeof x**
`undefined`	`"undefined"`
`null`	`"object"`
`true` **или** `false`	`"boolean"`
любое число или `NaN`	`"number"`
любая строка	`"string"`
любая функция	`"function"`
любой объект базового языка, не являющийся функцией	`"object"`
любой объект среды выполнения	**Строка, содержимое которой зависит от реализации, но не** `"undefined"`, `"boolean"`, `"number"` **или** `"string"`.

Оператор `typeof` может применяться, например, в таких выражениях:

```
(typeof value == "string") ? "'" + value + "'" : value
```

Оператор `typeof` можно также использовать в инструкции `switch` (раздел 5.4.3). Обратите внимание, что операнд оператора `typeof` можно заключить в скобки, что делает оператор `typeof` более похожим на имя функции, а не на ключевое слово или оператор:

```
typeof(i)
```

Обратите внимание, что для значения `null` оператор `typeof` возвращает строку «object». Если вам потребуется отличать `null` от других объектов, добавьте проверку для этого спецслучая. Для объектов, определяемых средой выполнения, оператор `typeof` может возвращать строку, отличную от «object». Однако на практике большинство таких объектов в клиентском JavaScript имеют тип «object».

Для всех объектных типов и типов массивов результатом оператора `typeof` является строка «object», поэтому он может быть полезен только для определения принадлежности значения к объектному или к простому типу. Чтобы отличить один класс объектов от другого, следует использовать другие инструменты, такие как оператор `instanceof` (раздел 4.9.4), атрибут *class* (раздел 6.8.2) или свойство `constructor` (разделы 6.8.1 и 9.2.2).

Несмотря на то что функции в JavaScript также являются разновидностью объектов, оператор `typeof` отличает функции, потому что они имеют собственные возвращаемые значения. В JavaScript имеется тонкое отличие между функциями и «вызываемыми объектами». Функции могут вызываться, но точно так же можно создать вызываемый объект – который может вызываться подобно функции, – не являющийся настоящей функцией. В спецификации ECMAScript 3 говорится, что оператор `typeof` должен возвращать строку «function» для всех объектов базового языка, которые могут вызываться. Спецификация ECMAScript 5 расширяет это требование и требует, чтобы оператор `typeof` возвращал строку «function» для всех вызываемых объектов, будь то объекты базового языка или среды выполнения. Большинство производителей броузеров для реализации методов своих объектов среды выполнения используют обычные объекты-функции базового языка JavaScript. Однако корпорация Microsoft для реализации своих клиентских методов всегда использовала собственные вызываемые объекты, вследствие чего в версиях до IE9 оператор `typeof` возвращает строку «object» для них, хотя они ведут себя как функции. В версии IE9 клиентские методы были реализованы как обычные объекты-функции базового языка. Подробнее об отличиях между истинными функциями и вызываемыми объектами рассказывается в разделе 8.7.7.

4.13.3. Оператор delete

Унарный оператор `delete` выполняет попытку удалить свойство объекта или элемент массива, определяемый операндом.[1] Подобно операторам присваивания, инкремента и декремента, оператор `delete` обычно используется ради побочного эффекта, выражающегося в удалении свойства, а не ради возвращаемого значения. Ниже приводятся несколько примеров его использования:

```
var o = {x: 1, y: 2}; // Определить объект
delete o.x;           // Удалить одно из его свойств
"x" in o              // => false: свойство больше не существует

var a = [1,2,3];      // Создать массив
delete a[2];          // Удалить последний элемент массива
2 in a               // => false: второй элемент больше не существует
a.length            // => 3: обратите внимание, что длина массива при этом не изменилась
```

Внимание: удаленное свойство или элемент массива не просто получает значение `undefined`. После удаления свойства оно прекращает свое существование. Попытка прочитать значение несуществующего свойства возвратит значение `undefined`, но вы можете проверить фактическое наличие свойства с помощью оператора `in` (раздел 4.9.3). Операция удаления элемента массива оставляет в массиве «дырку» и не изменяет длину массива. В результате получается *разреженный* массив.

Оператор `delete` требует, чтобы его операнд был левосторонним выражением. Если операнд не является левосторонним выражением, оператор не будет выполнять никаких действий и вернет значение `true`. В противном случае `delete` попытается удалить указанное левостороннее выражение. В случае успешного удаления значения левостороннего выражения оператор `delete` вернет значение `true`. Не все свойства могут быть удалены: некоторые встроенные свойства из базового и клиентского языков JavaScript устойчивы к операции удаления. Точно так же не могут быть удалены пользовательские переменные, объявленные с помощью инструкции `var`. Кроме того, невозможно удалить функции, объявленные с помощью инструкции `function`, а также объявленные параметры функций.

В строгом режиме, определяемом стандартом ECMAScript 5, оператор `delete` возбуждает исключение `SyntaxError`, если его операндом является неквалифицированный идентификатор, такой как имя переменной, функции или параметра функции: он может оперировать только операндами, которые являются выражениями обращения к свойству (раздел 4.4). Кроме того, строгий режим определяет, что оператор `delete` должен возбуждать исключение `TypeError`, если запрошено удаление ненастраиваемого свойства (раздел 6.7). В обычном режиме в таких случаях исключение не возбуждается, и оператор `delete` просто возвращает `false`, чтобы показать, что операнд не был удален.

---

[1] Программистам на C++ следует обратить внимание, что `delete` в JavaScript совершенно не похож на `delete` в C++. В JavaScript освобождение памяти выполняется сборщиком мусора автоматически, и беспокоиться о явном освобождении памяти не надо. Поэтому в операторе `delete` в стиле C++, удаляющем объекты без остатка, нет необходимости.

Ниже приводится несколько примеров использования оператора `delete`:

```
var o = {x:1, y:2}; // Определить переменную; инициализировать ее объектом
delete o.x;         // Удалить одно из свойств объекта; вернет true
typeof o.x;         // Свойство не существует; вернет "undefined"
delete o.x;         // Удалить несуществующее свойство; вернет true
delete o;           // Объявленную переменную удалить нельзя; вернет false
                    // В строгом режиме возбудит исключение.
delete 1;           // Аргумент не является левосторонним выражением; вернет true
this.x = 1;         // Определить свойство глобального объекта без var
delete x;           // Удалить: вернет true при выполнении в нестрогом режиме; в строгом
                    // режиме возбудит исключение. Используйте 'delete this.x' взамен.
x;                  // Ошибка времени выполнения: переменная x не определена
```

С оператором `delete` мы снова встретимся в разделе 6.3.

4.13.4. Оператор void

Унарный оператор `void` указывается перед своим единственным операндом любого типа. Этот оператор редко используется и имеет необычное действие: он вычисляет значение операнда, затем отбрасывает его и возвращает `undefined`. Поскольку значение операнда отбрасывается, использовать оператор `void` имеет смысл только ради побочных эффектов, которые дает вычисление операнда.

Чаще всего этот оператор применяется в клиентском JavaScript, в адресах URL вида `JavaScript:`, где он позволяет вычислить выражение ради его побочных действий, не отображая в броузере вычисленное значение. Например, оператор `void` можно использовать в HTML-теге `<a>`:

```
<a href="javascript:void window.open();">Открыть новое окно</a>
```

Эта разметка HTML была бы более очевидна, если бы вместо URL `javascript:` применялся обработчик события `onclick`, где в использовании оператора `void` нет никакой необходимости.

4.13.5. Оператор «запятая» (,)

Оператор «запятая» (`,`) является двухместным оператором и может принимать операнды любого типа. Он вычисляет свой левый операнд, вычисляет свой правый операнд и возвращает значение правого операнда. То есть следующая строка:

```
i=0, j=1, k=2;
```

вернет значение 2 и практически эквивалентна строке:

```
i = 0; j = 1; k = 2;
```

Выражение слева вычисляется всегда, но его значение отбрасывается, поэтому применять оператор запятая имеет смысл только ради побочного эффекта левого операнда. Единственным типичным применением оператора запятая является его использование в циклах `for` (раздел 5.5.3) с несколькими переменными цикла:

```
// Первая запятая ниже является частью синтаксиса инструкции var
// Вторая запятая является оператором: она позволяет внедрить 2 выражения (i++ и j--)
// в инструкцию (цикл for), которая ожидает 1 выражение.
for(var i=0,j=10; i < j; i++,j--)
    console.log(i+j);
```

5

Инструкции

В главе 4 выражения были названы фразами языка JavaScript. По аналогии инструкции можно считать предложениями на языке JavaScript, или командами. Как в обычном языке предложения завершаются и отделяются друг от друга точками, так же и инструкции JavaScript завершаются точками с запятой (раздел 2.5). Выражения *вычисляются* и возвращают значение, а инструкции *выполняются*, чтобы что-то происходило.

Чтобы «что-то происходило», можно вычислить выражение, имеющее побочные эффекты. Выражения с побочными эффектами, такие как присваивание и вызовы функций, могут играть роль самостоятельных инструкций – при таком использовании их обычно называют *инструкциями-выражениями*. Похожую категорию инструкций образуют *инструкции-объявления*, которые объявляют новые переменные и определяют новые функции.

Программы на языке JavaScript представляют собой не более чем последовательности выполняемых инструкций. По умолчанию интерпретатор JavaScript выполняет эти инструкции одну за другой в порядке их следования. Другой способ сделать так, чтобы «что-то происходило», заключается в том, чтобы влиять на этот порядок выполнения по умолчанию, для чего в языке JavaScript имеется несколько инструкций, или управляющих конструкций, специально предназначенных для этого:

- *Условные инструкции*, такие как `if` и `switch`, которые заставляют интерпретатор JavaScript выполнять или пропускать другие инструкции в зависимости от значения выражения.
- *Инструкции циклов*, такие как `while` и `for`, которые многократно выполняют другие инструкции.
- *Инструкции переходов*, такие как `break`, `return` и `throw`, которые заставляют интерпретатор выполнить переход в другую часть программы.

В разделах, следующих далее, описываются различные инструкции языка JavaScript и их синтаксис. Краткая сводка синтаксиса инструкций приводится в табл. 5.1, в конце главы. Программа на языке JavaScript – это просто последовательность инструкций, разделенных точками с запятой, поэтому, познакомившись с инструкциями JavaScript, вы сможете писать программы на этом языке.

5.1. Инструкции-выражения

Простейший вид инструкций в JavaScript – это выражения, имеющие побочные эффекты. (Загляните в раздел 5.7.3, где описывается инструкция-выражение, не имеющая побочных эффектов.) Инструкции такого рода мы рассматривали в главе 4. Основной категорией инструкций-выражений являются инструкции присваивания. Например:

```
greeting = "Hello " + name;
i *= 3;
```

Операторы инкремента и декремента, `++` и `--` схожи с инструкциями присваивания. Их побочным эффектом является изменение значения переменной, как при выполнении присваивания:

```
counter++;
```

Оператор `delete` имеет важный побочный эффект – он удаляет свойство объекта. Поэтому он почти всегда применяется как инструкция, а не как часть более сложного выражения:

```
delete o.x;
```

Вызовы функций – еще одна большая категория инструкций-выражений. Например:

```
alert(greeting);
window.close();
```

Эти вызовы клиентских функций являются выражениями, однако они имеют побочный эффект, заключающийся в воздействии на веб-броузер, поэтому также могут использоваться в качестве инструкций. Если функция не имеет каких-либо побочных эффектов, нет смысла вызывать ее, если только она не является частью более сложного выражения или инструкции присваивания. Например, никто не станет просто вычислять косинус и отбрасывать результат:

```
Math.cos(x);
```

Наоборот, надо вычислить значение и присвоить его переменной для дальнейшего использования:

```
cx = Math.cos(x);
```

Обратите внимание, что каждая строка в этих примерах завершается точкой с запятой.

5.2. Составные и пустые инструкции

Подобно оператору запятой (раздел 4.13.5), объединяющему несколько выражений в одно выражение, *блок инструкций* позволяет объединить несколько инструкций в одну *составную инструкцию*. Блок инструкций – это просто последовательность инструкций, заключенная в фигурные скобки. Таким образом, следующие строки рассматриваются как одна инструкция и могут использоваться везде, где интерпретатор JavaScript требует наличия единственной инструкции:

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

Здесь есть несколько аспектов, на которые следует обратить внимание. Во-первых, составная инструкция *не* завершается точкой с запятой. Отдельные инструкции внутри блока завершаются точками с запятой, однако сам блок – нет. Во-вторых, строки внутри блока оформлены с отступами относительно фигурных скобок, окружающих их. Это не является обязательным требованием, но подобное оформление программного кода упрощает его чтение и понимание. Наконец, напомню, что в языке JavaScript не поддерживается область видимости блока, поэтому переменные, объявленные внутри блока, не являются частными по отношению к этому блоку (подробности смотрите в разделе 3.10.1).

Объединение инструкций в более крупные блоки инструкций используется в языке JavaScript повсеместно. Подобно тому как выражения часто включают другие подвыражения, многие инструкции JavaScript могут содержать другие инструкции. Формальный синтаксис языка JavaScript обычно позволяет использовать не более одной подынструкции. Например, синтаксис инструкции цикла `while` включает единственную подынструкцию, которая служит телом цикла. Блоки инструкций позволяют помещать любое количество инструкций там, где требуется наличие единственной подынструкции.

Составные инструкции позволяют использовать множество инструкций там, где синтаксис JavaScript допускает только одну инструкцию. *Пустая инструкция* действует противоположным образом: она позволяет не вставлять инструкции там, где они необходимы. Пустая инструкция имеет следующий вид:

Встретив пустую инструкцию, интерпретатор JavaScript не выполняет никаких действий. Пустая инструкция может оказаться полезной, когда требуется создать цикл с пустым телом. Взгляните на следующий цикл `for` (циклы `for` будут рассматриваться в разделе 5.5.3):

```
// Инициализировать массив a
for(i = 0; i < a.length; a[i++] = 0) ;
```

В этом цикле вся работа выполняется выражением `a[i++] = 0`, и тело цикла здесь не требуется. Однако синтаксис JavaScript требует, чтобы цикл имел тело, поэтому здесь использована пустая инструкция – просто точка с запятой.

Обратите внимание, что ошибочное добавление точки с запятой после закрывающей круглой скобки в инструкции `for`, `while` или `if` может вызывать появление досадных ошибок, которые сложно обнаружить. Например, следующий фрагмент наверняка будет делать не совсем то, что предполагал автор:

```
if ((a == 0) || (b == 0));    // Ой! Эта строка ничего не делает...
    o = null;                 // а эта будет выполняться всегда.
```

Если вы собираетесь намеренно использовать пустую инструкцию, нелишним будет добавить комментарий, поясняющий ваши намерения. Например:

```
for(i = 0; i < a.length; a[i++] = 0) /* пустое тело цикла */ ;
```

5.3. Инструкции-объявления

Инструкции `var` и `function` являются *инструкциями-объявлениями* – они объявляют, или определяют, переменные и функции. Эти инструкции определяют идентификаторы (имена переменных и функций), которые могут использоваться повсюду в программе, и присваивают значения этим идентификаторам. Инструкции-объявления сами ничего особенного не делают, но, создавая переменные и функции, они в значительной степени определяют значение других инструкций в программе.

В подразделах, следующих ниже, описываются инструкции `var` и `function`, но они не дают исчерпывающего описания переменных и функций. Более подробная информация о переменных приводится в разделах 3.9 и 3.10, а полное описание функций – в главе 8.

5.3.1. Инструкция var

Инструкция `var` позволяет явно объявить одну или несколько переменных. Инструкция имеет следующий синтаксис:

```
var имя_1 [ = значение_1] [ ,..., имя_n [= значение_n]]
```

За ключевым словом `var` следует список объявляемых переменных через запятую; каждая переменная в списке может иметь специальное выражение-инициализатор, определяющее ее начальное значение. Например:

```
var i;                                    // Одна простая переменная
var j = 0;                                // Одна переменная, одно значение
var p, q;                                 // Две переменные
var greeting = "hello" + name;            // Сложный инициализатор
var x = 2.34, y = Math.cos(0.75), r, theta; // Множество переменных
var x = 2, y = x*x;                       // Вторая переменная использует первую
var x = 2,                                // Множество переменных...
    f = function(x) { return x*x },       // каждая определяется
    y = f(x);                             // в отдельной строке
```

Если инструкция `var` находится внутри тела функции, она определяет локальные переменные, видимые только внутри этой функции. При использовании в программном коде верхнего уровня инструкция `var` определяет глобальные переменные, видимые из любого места в программе. Как отмечалось в разделе 3.10.2, глобальные переменные являются свойствами глобального объекта. Однако, в отличие от других глобальных свойств, свойства, созданные с помощью инструкции `var`, нельзя удалить.

Если в инструкции `var` начальное значение переменной не указано, то переменная определяется, однако ее начальное значение остается неопределенным (`undefined`). Как описывалось в разделе 3.10.1, переменные определены во всем сценарии или в функции, где они были объявлены, – их объявления «поднимаются» в начало сценария или функции. Однако инициализация переменной производится в той точке программы, где находится инструкция `var`, а до этого переменная имеет значение `undefined`.

Обратите внимание, что инструкция `var` может также являться частью циклов `for` и `for/in`. (Объявления этих переменных так же поднимаются в начало сценария или функции, как и объявления других переменных вне цикла.) Ниже повторно приводятся примеры из раздела 3.9:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var i in o) console.log(i);
```

Отметьте также, что допускается несколько раз объявлять одну и ту же переменную.

5.3.2. Инструкция function

Ключевое слово `function` в языке JavaScript используется для определения функций. В разделе 4.3 мы уже встречались с выражением определения функции. Но функции можно также определять в форме инструкций. Взгляните на следующие две функции:

```
var f = function(x) { return x+1; } // Выражение присваивается переменной
function f(x) { return x+1; }       // Инструкция включает имя переменной
```

Объявление функции в форме инструкции имеет следующий синтаксис:

```
function имя_функции([арг1 [,арг2 [..., аргn]]]) {
    инструкции
}
```

Здесь *имя_функции* – это идентификатор, определяющий имя объявляемой функции. За именем функции следует заключенный в скобки список имен аргументов, разделенных запятыми. Эти идентификаторы могут использоваться в теле функции для ссылки на значения аргументов, переданных при вызове функции.

Тело функции состоит из произвольного числа JavaScript-инструкций, заключенных в фигурные скобки. Эти инструкции не выполняются при определении функции. Они просто связываются с новым объектом функции для выполнения при ее вызове. Обратите внимание, что фигурные скобки являются обязательной частью инструкции `function`. В отличие от блоков инструкций в циклах `while` и в других конструкциях, тело функции требует наличия фигурных скобок, даже если оно состоит только из одной инструкции.

Ниже приводится несколько примеров определений функций:

```
function hypotenuse(x, y) {
    return Math.sqrt(x*x + y*y);   // Инструкция return описывается далее
}

function factorial(n) {            // Рекурсивная функция
    if (n <= 1) return 1;
        return n * factorial(n - 1);
}
```

Инструкции объявления функций могут находиться в JavaScript-коде верхнего уровня или быть вложенными в определения других функций, но только на «верхнем уровне», т. е. объявления функций не могут находиться внутри инструкций `if`, циклов `while` или любых других конструкций. Из-за такого ограничения, накладываемого на объявления функций, спецификация ECMAScript не относит объявления функций к истинным инструкциям. Некоторые реализации JavaScript позволяют вставлять объявления функций в любые инструкции, но разные реализации по-разному обрабатывают эти случаи, поэтому включение объявлений функций в другие инструкции снижает переносимость программ.

Инструкция объявления функции отличается от выражения тем, что она включает имя функции. Обе формы создают новый объект функции, но инструкция объявления функции при этом объявляет имя функции – переменную, которой присваивается объект функции. Подобно переменным, объявляемым с помощью инструкции `var`, объявления функций, созданные с помощью инструкции `function`, неявно «поднимаются» в начало содержащего их сценария или функции, поэтому они видимы из любого места в сценарии или функции. Однако при использовании инструкции `var` поднимается только объявление переменной, а инициализация остается там, куда ее поместил программист. В случае же с инструкцией `function` поднимается не только имя функции, но и ее тело: все функции в сценарии или все функции, вложенные в функцию, будут объявлены до того, как начнется выполнение программного кода. Это означает, что функцию можно вызвать еще до того, как она будет объявлена.

Подобно инструкции `var`, инструкции объявления функций создают переменные, которые невозможно удалить. Однако эти переменные доступны не только для чтения – им можно присвоить другие значения.

5.4. Условные инструкции

Условные инструкции позволяют пропустить или выполнить другие инструкции в зависимости от значения указанного выражения. Эти инструкции являются точками принятия решений в программе, и иногда их также называют инструкциями «ветвления». Если представить, что программа – это дорога, а интерпретатор JavaScript – путешественник, идущий по ней, то условные инструкции можно представить как перекрестки, где программный код разветвляется на две или более дорог, и на таких перекрестках интерпретатор должен выбирать, по какой дороге двигаться дальше.

В подразделах ниже описывается основная условная инструкция языка JavaScript – инструкция `if/else`, а также более сложная инструкция `switch`, позволяющая создавать множество ответвлений.

5.4.1. Инструкция if

Инструкция `if` – это базовая управляющая инструкция, позволяющая интерпретатору JavaScript принимать решения или, точнее, выполнять инструкции в зависимости от условий. Инструкция имеет две формы. Первая:

```
if (выражение)
    инструкция
```

В этой форме сначала вычисляется *выражение*. Если полученный результат является истинным, то *инструкция* выполняется. Если *выражение* возвращает ложное значение, то *инструкция* не выполняется. (Определения истинных и ложных значений приводятся в разделе 3.3.) Например:

```
if (username == null)       // Если переменная username равна null или undefined,
    username = "John Doe";  // определить ее
```

Аналогично:

```
// Если переменная username равна null, undefined, 0, "" или NaN,
// присвоить ей новое значение.
if (!username) username = "John Doe";
```

Обратите внимание, что скобки вокруг условного выражения являются обязательной частью синтаксиса инструкции `if`.

Синтаксис языка JavaScript позволяет вставить только одну инструкцию после инструкции `if` и выражения в круглых скобках, однако одиночную инструкцию всегда можно заменить блоком инструкций. Поэтому инструкция `if` может выглядеть так:

```
if (!address) {
    address = "";
    message = "Пожалуйста, укажите почтовый адрес.";
}
```

Вторая форма инструкции `if` вводит конструкцию `else`, выполняемую в тех случаях, когда *выражение* возвращает ложное значение. Ее синтаксис:

```
if (выражение)
    инструкция1
else
    инструкция2
```

Эта форма инструкции выполняет *инструкцию1*, если *выражение* возвращает истинное значение, и *инструкцию2*, если *выражение* возвращает ложное значение. Например:

```
if (n == 1)
    console.log("Получено 1 новое сообщение.");
else
    console.log("Получено " + n + " новых сообщений.");
```

При наличии вложенных инструкций `if` с блоками `else` требуется некоторая осторожность – необходимо гарантировать, что `else` относится к соответствующей ей инструкции `if`. Взгляните на следующие строки:

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        console.log("i равно k");
else
    console.log("i не равно j"); // НЕПРАВИЛЬНО!!
```

В этом примере внутренняя инструкция `if` является единственной инструкцией, вложенной во внешнюю инструкцию `if`. К сожалению, неясно (если исключить подсказку, которую дают отступы), к какой инструкции `if` относится блок `else`. А отступы в этом примере выставлены неправильно, потому что в действительности интерпретатор JavaScript интерпретирует предыдущий пример так:

```
if (i == j) {
    if (j == k)
        console.log("i равно k");
    else
        console.log("i не равно j"); // Вот как!
}
```

Согласно правилам JavaScript (и большинства других языков программирования), конструкция `else` является частью ближайшей к ней инструкции `if`. Чтобы сделать этот пример менее двусмысленным и более легким для чтения, понимания, сопровождения и отладки, надо поставить фигурные скобки:

```
if (i == j) {
    if (j == k) {
        console.log("i равно k");
    }
}
else { // Вот какая разница возникает из-за добавления фигурных скобок!
    console.log("i не равно j");
}
```

Хотя этот стиль и не используется в данной книге, тем не менее многие программисты заключают тела инструкций `if` и `else` (а также других составных инструкций, таких как циклы `while`) в фигурные скобки, даже когда тело состоит только из одной инструкции. Последовательное применение этого правила поможет избежать неприятностей, подобных только что описанной.

5.4.2. Инструкция else if

Инструкция `if/else` вычисляет значение выражения и выполняет тот или иной фрагмент программного кода, а зависимости от результата. Но что если требуется выполнить один из многих фрагментов? Возможный способ сделать это состоит в применении инструкции `else if`. Формально она не является самостоятельной инструкцией JavaScript; это лишь распространенный стиль программирования, заключающийся в применении повторяющихся инструкций `if/else`:

```
if (n == 1) {
    // Выполнить блок 1
}
else if (n == 2) {
    // Выполнить блок 2
}
else if (n == 3) {
    // Выполнить блок 3
}
else {
    // Если ни одна из предыдущих инструкций else не была выполнена, выполнить блок 4
}
```

В этом фрагменте нет ничего особенного. Это просто последовательность инструкций `if`, где каждая инструкция `if` является частью конструкции `else` предыдущей инструкции. Стиль `else if` предпочтительнее и понятнее записи в синтаксически эквивалентной форме, полностью показывающей вложенность инструкций:

```
if (n == 1) {
   // Выполнить блок 1
}
else {
   if (n == 2) {
      // Выполнить блок 2
   }
   else {
      if (n == 3) {
         // Выполнить блок 3
      }
      else {
         // Если ни одна из предыдущих инструкций else
         // не была выполнена, выполнить блок 4
      }
   }
}
```

5.4.3. Инструкция switch

Инструкция `if` создает ветвление в потоке выполнения программы, а многопозиционное ветвление можно реализовать посредством нескольких инструкций `else if`. Однако это не всегда наилучшее решение, особенно если все ветви зависят от значения одного и того же выражения. В этом случае расточительно повторно вычислять значение одного и того же выражения в нескольких инструкциях `if`.

Инструкция `switch` предназначена именно для таких ситуаций. За ключевым словом `switch` следует выражение в скобках и блок кода в фигурных скобках:

```
switch(выражение) {
   инструкции
}
```

Однако полный синтаксис инструкции `switch` более сложен, чем показано здесь. Различные места в блоке помечены ключевым словом `case`, за которым следует выражение и символ двоеточия. Ключевое слово `case` напоминает инструкцию с меткой за исключением того, что оно связывает инструкцию с выражением, а не с именем. Когда выполняется инструкция `switch`, она вычисляет значение *выражения*, а затем ищет метку `case`, соответствующую этому значению (соответствие определяется с помощью оператора идентичности `===`). Если метка найдена, выполняется блок кода, начиная с первой инструкции, следующей за меткой `case`. Если метка `case` с соответствующим значением не найдена, выполнение начинается с первой инструкции, следующей за специальной меткой `default:`. Если метка `default:` отсутствует, блок инструкции `switch` пропускается целиком.

Работу инструкции `switch` сложно объяснить на словах, гораздо понятнее выглядит объяснение на примере. Следующая инструкция `switch` эквивалентна повторяющимся инструкциям `if/else`, показанным в предыдущем разделе:

```
switch(n) {
   case 1:     // Выполняется, если n === 1
      // Выполнить блок 1.
      break; // Здесь остановиться
   case 2:     // Выполняется, если n === 2
```

```
        // Выполнить блок 2.
        break; // Здесь остановиться
    case 3:    // Выполняется, если n === 3
        // Выполнить блок 3.
        break; // Здесь остановиться
    default:   // Если все остальное не подходит...
        // Выполнить блок 4.
        break; // Здесь остановиться
}
```

Обратите внимание на ключевое слово `break` в конце каждого блока `case`. Инструкция `break`, описываемая далее в этой главе, приводит к передаче управления в конец инструкции `switch` и продолжению выполнения инструкций, следующих далее. Конструкции `case` в инструкции `switch` задают только *начальную точку* выполняемого программного кода, но не задают никаких конечных точек. В случае отсутствия инструкций `break` инструкция `switch` начнет выполнение блока кода с меткой `case`, соответствующей значению *выражения*, и продолжит выполнение инструкций до тех пор, пока не дойдет до конца блока. В редких случаях это полезно для написания программного кода, который переходит от одной метки `case` к следующей, но в 99% случаев следует аккуратно завершать каждый блок `case` инструкцией `break`. (При использовании `switch` внутри функции вместо `break` можно использовать инструкцию `return`. Обе эти инструкции служат для завершения работы инструкции `switch` и предотвращения перехода к следующей метке `case`.)

Ниже приводится более практичный пример использования инструкции `switch`; он преобразует значение в строку способом, зависящим от типа значения:

```
function convert(x) {
    switch(typeof x) {
        case 'number':               // Преобразовать число в шестнадцатеричное целое
            return x.toString(16);
        case 'string':               // Вернуть строку, заключенную в кавычки
            return '"' + x + '"';
        default:                     // Любой другой тип преобразуется обычным способом
            return x.toString()
    }
}
```

Обратите внимание, что в двух предыдущих примерах за ключевыми словами `case` следовали числа или строковые литералы. Именно так инструкция `switch` чаще всего используется на практике, но стандарт ECMAScript позволяет указывать после `case` произвольные выражения.

Инструкция `switch` сначала вычисляет выражение после ключевого слова `switch`, а затем выражения `case` в том порядке, в котором они указаны, пока не будет найдено совпадающее значение.[1] Факт совпадения определяется с помощью оператора идентичности `===`, а не с помощью оператора равенства `==`, поэтому выражения должны совпадать без какого-либо преобразования типов.

---

[1] Тот факт, что выражения в метках `case` вычисляются во время выполнения программы, существенно отличает инструкцию `switch` в языке JavaScript (и делает ее менее эффективной) от инструкции `switch` в C, C++ и Java. В этих языках выражения `case` должны быть константами, вычисляемыми на этапе компиляции, и иметь один тот же тип. Кроме того, инструкция `switch` в этих языках часто может быть реализована с использованием высокоэффективной *таблицы переходов*.

Поскольку при каждом выполнении инструкции `switch` вычисляются не все выражения `case`, следует избегать использования выражений `case`, имеющих побочные эффекты, такие как вызовы функций и присваивания. Безопаснее всего ограничиваться в выражениях `case` константными выражениями.

Как объяснялось ранее, если ни одно из выражений `case` не соответствует выражению `switch`, инструкция `switch` начинает выполнение с инструкции с меткой `default:`. Если метка `default:` отсутствует, тело инструкции `switch` полностью пропускается. Обратите внимание, что в предыдущих примерах метка `default:` указана в конце тела инструкции `switch` после всех меток `case`. Это логичное и обычное место для нее, но на самом деле она может располагаться в любом месте внутри инструкции `switch`.

5.5. Циклы

Чтобы понять действие условных инструкций, мы предлагали представить их в виде разветвлений на дороге, по которой двигается интерпретатор JavaScript. *Инструкции циклов* можно представить как разворот на дороге, возвращающий обратно, который заставляет интерпретатор многократно проходить через один и тот же участок программного кода. В языке JavaScript имеется четыре инструкции циклов: `while`, `do/while`, `for` и `for/in`. Каждому из них посвящен один из следующих подразделов. Одно из обычных применений инструкций циклов – обход элементов массива. Эта разновидность циклов подробно обсуждается в разделе 7.6, где также рассматриваются специальные методы итераций класса `Array`.

5.5.1. Инструкция while

Инструкция `if` является базовой условной инструкцией в языке JavaScript, а базовой инструкцией циклов для JavaScript можно считать инструкцию `while`. Она имеет следующий синтаксис:

```
while (выражение)
   инструкция
```

Инструкция `while` начинает работу с вычисления *выражения*. Если это выражение имеет ложное значение, интерпретатор пропускает *инструкцию*, составляющую тело цикла, и переходит к следующей инструкции в программе. Если *выражение* имеет истинное значение, то выполняется *инструкция*, образующая тело цикла, затем управление передается в начало цикла и *выражение* вычисляется снова. Иными словами, интерпретатор снова и снова выполняет *инструкцию* тела цикла, *пока* (while) значение *выражения* остается истинным. Обратите внимание, что имеется возможность организовать бесконечный цикл с помощью синтаксиса `while(true)`.

Обычно не требуется, чтобы интерпретатор JavaScript снова и снова выполнял одну и ту же операцию. Почти в каждом цикле с каждой *итерацией* цикла одна или несколько переменных изменяют свои значения. Поскольку переменная меняется, действия, которые выполняет *инструкция*, при каждом проходе тела цикла могут отличаться. Кроме того, если изменяемая переменная (или переменные) присутствует в *выражении*, значение выражения может меняться при каждом проходе цикла. Это важно, т. к. в противном случае выражение, значение которого было истинным, никогда не изменится и цикл никогда не завершится! Ниже приводится пример цикла `while`, который выводит числа от 0 до 9:

```
var count = 0;
while (count < 10) {
    console.log(count);
    count++;
}
```

Как видите, в начале переменной `count` присваивается значение 0, а затем ее значение увеличивается каждый раз, когда выполняется тело цикла. После того как цикл будет выполнен 10 раз, выражение вернет `false` (т. е. переменная `count` уже не меньше 10), инструкция `while` завершится и интерпретатор перейдет к следующей инструкции в программе. Большинство циклов имеют переменные-счетчики, аналогичные `count`. Чаще всего в качестве счетчиков цикла выступают переменные с именами `i`, `j` и `k`, хотя для того чтобы сделать программный код более понятным, следует давать счетчикам более наглядные имена.

5.5.2. Инструкция do/while

Цикл `do/while` во многом похож на цикл `while`, за исключением того, что выражение цикла проверяется в конце, а не в начале. Это значит, что тело цикла всегда выполняется как минимум один раз. Эта инструкция имеет следующий синтаксис:

```
do
    инструкция
while (выражение);
```

Цикл `do/while` используется реже, чем родственный ему цикл `while`. Дело в том, что на практике ситуация, когда вы заранее уверены, что потребуется хотя бы один раз выполнить тело цикла, несколько необычна. Ниже приводится пример использования цикла `do/while`:

```
function printArray(a) {
    var len = a.length, i = 0;
    if (len == 0)
        console.log("Пустой массив");
    else {
        do {
            console.log(a[i]);
        } while (++i < len);
    }
}
```

Между циклом `do/while` и обычным циклом `while` имеется два отличия. Во-первых, цикл `do` требует как ключевого слова `do` (для отметки начала цикла), так и ключевого слова `while` (для отметки конца цикла и указания условия). Во-вторых, в отличие от цикла `while`, цикл `do` завершается точкой с запятой. Цикл `while` необязательно завершать точкой с запятой, если тело цикла заключено в фигурные скобки.

5.5.3. Инструкция for

Инструкция `for` представляет собой конструкцию цикла, которая часто оказывается более удобной, чем инструкция `while`. Инструкция `for` упрощает конструирование циклов, следующих шаблону, общему для большинства циклов. Большинство циклов имеют некоторую переменную-счетчик. Эта переменная инициализируется перед началом цикла и проверяется перед каждой итерацией. Наконец, переменная-счетчик инкрементируется или изменяется каким-либо другим образом в конце тела цикла, непосредственно перед повторной проверкой переменной. Инициализация, проверка и обновление – это три ключевых операции, выполняемых с переменной цикла. Инструкция `for` делает эти три шага явной частью синтаксиса цикла:

```
for(инициализация; проверка; инкремент)
    инструкция
```

Инициализация, *проверка* и *инкремент* – это три выражения (разделенных точкой с запятой), которые ответственны за инициализацию, проверку и увеличение переменной цикла. Расположение их в первой строке цикла упрощает понимание того, что делает цикл `for`, и не позволяет забыть инициализировать или увеличить переменную цикла.

Проще всего объяснить работу цикла `for`, показав эквивалентный ему цикл `while`:[1]

```
инициализация;
while(проверка) {
    инструкция
    инкремент;
}
```

Другими словами, выражение *инициализации* вычисляется один раз перед началом цикла. Это выражение, как правило, является выражением с побочными эффектами (обычно присваиванием). В JavaScript также допускается, чтобы выражение *инициализации* было инструкцией объявления переменной `var`, поэтому можно одновременно объявить и инициализировать счетчик цикла. Выражение *проверки* вычисляется перед каждой итерацией и определяет, будет ли выполняться тело цикла. Если результатом проверки является истинное значение, выполняется *инструкция*, являющаяся телом цикла. В конце цикла вычисляется выражение *инкремент*. Чтобы использование этого выражения имело смысл, оно должно быть выражением с побочными эффектами. Обычно это либо выражение присваивания, либо выражение, использующее оператор ++ или --.

Вывести числа от 0 до 9 можно также с помощью цикла `for`, как показано ниже. В противовес эквивалентному циклу `while`, показанному в предыдущем разделе:

```
for(var count = 0; count < 10; count++)
    console.log(count);
```

Конечно, циклы могут быть значительно более сложными, чем в этих простых примерах, и иногда в каждой итерации цикла изменяется несколько переменных. Эта ситуация – единственный случай в JavaScript, когда часто применяется оператор «запятая» – он позволяет объединить несколько выражений инициализации и инкрементирования в одно выражение, подходящее для использования в цикле `for`:

---

[1] Как мы увидим, когда будем знакомиться с инструкцией `continue` в разделе 5.6.3, этот цикл `while` не является точным эквивалентом цикла `for`.

```
var i, j
for(i = 0, j = 10; i < 10; i++, j--)
    sum += i * j;
```

Во всех наших примерах циклов, представленных до сих пор, переменная цикла содержала число. Это достаточно распространенная, но не обязательная практика. В следующем примере цикл `for` используется для обхода связанного списка структур данных и получения последнего объекта в списке (например, первого объекта, который не имеет свойства `next`):

```
function tail(o) {                          // Возвращает последний элемент в списке o
    for(; o.next; o = o.next) /* пустое */ ; // Выполнять обход, пока o.next
        return o;                           // является истинным значением
}
```

Обратите внимание на отсутствие выражения инициализации в примере выше. Любое из трех выражений цикла `for` может быть опущено, но две точки с запятой являются обязательными. Если опустить выражение *проверки*, цикл будет повторяться вечно, и форма записи `for(;;)` является еще одним способом написать бесконечный цикл, подобно `while(true)`.

5.5.4. Инструкция for/in

Инструкция цикла `for/in` использует ключевое слово `for`, но она в корне отличается от инструкции обычного цикла `for`. Цикл `for/in` имеет следующий синтаксис:

```
for (переменная in объект)
    инструкция
```

В качестве *переменной* здесь обычно используется имя переменной, но точно так же можно использовать любое выражение, возвращающее левостороннее выражение (раздел 4.7.3), или инструкцию `var`, объявляющую единственную переменную, – практически все, что может находиться слева от оператора присваивания. Параметр *объект* – это выражение, возвращающее объект. И как обычно, *инструкция* – это инструкция или блок инструкций, образующих тело цикла.

Для обхода элементов массива естественно использовать обычный цикл `for`:

```
for(var i = 0; i < a.length; i++)   // Присваивать индексы в массиве переменной i
    console.log(a[i]);              // Вывести значение каждого элемента массива
```

Инструкция `for/in` так же естественно позволяет выполнить обход свойств объекта.

```
for(var p in o)                     // Присваивать имена свойств объекта o переменной p
    console.log(o[p]);              // Вывести значение каждого свойства
```

Чтобы выполнить инструкцию `for/in`, интерпретатор JavaScript сначала вычисляет выражение *объект*. Если оно возвращает значение `null` или `undefined`, интерпретатор пропускает цикл и переходит к следующей инструкции.[1] Если выражение возвращает простое значение, оно преобразуется в эквивалентный объект-обертку (раздел 3.6). В противном случае выражение возвращает объект. Затем интерпретатор выполняет по одной итерации цикла для каждого перечислимого свойства объекта. Перед каждой итерацией интерпретатор вычисляет значение выражения *переменная* и присваивает ему имя свойства (строковое значение).

---

[1] Реализации, следующие стандарту ECMAScript 3, в этом случае могут возбуждать исключение `TypeError`.

Обратите внимание, что *переменная* в цикле `for/in` может быть любым выражением, возвращающим значение, которое можно использовать слева от оператора присваивания. Это выражение вычисляется в каждой итерации цикла, т. е. каждый раз оно может возвращать разные значения. Например, чтобы скопировать имена всех свойств объекта в массив, можно использовать следующий цикл:

```
var o = {x:1, y:2, z:3};
var a = []; var i = 0;
for(a[i++] in o) /* пустое тело цикла */;
```

Массивы в JavaScript – это просто специальный тип объектов, а индексы в массиве – свойства объекта, обход которых можно выполнить с помощью цикла `for/in`. Например, следующая инструкция перечислит индексы 0, 1 и 2 массива, объявленного выше:

```
for(i in a) console.log(i);
```

В действительности цикл `for/in` может совершать обход не по всем свойствам объекта, а только по *перечислимым* свойствам (раздел 6.7). Многочисленные встроенные методы, определяемые в базовом языке JavaScript, не являются перечислимыми. Например, все объекты имеют метод `toString()`, но цикл `for/in` не перечислит свойство `toString`. Кроме встроенных методов также не являются перечислимыми многие другие свойства встроенных объектов. При этом все свойства и методы, определяемые пользователем, являются перечислимыми. (Но в реализации, следующей стандарту ECMAScript 5, имеется возможность сделать их неперечислимыми, использовав прием, описанный в разделе 6.7.) Унаследованные свойства, которые были определены пользователем (раздел 6.2.2), также перечисляются циклом `for/in`.

Если в теле цикла `for/in` удалить свойство, которое еще не было перечислено, это свойство перечислено не будет. Если в теле цикла создать новые свойства, то обычно такие свойстве не будут перечислены. (Однако некоторые реализации могут перечислять унаследованные свойства, добавленные в ходе выполнения цикла.)

5.5.4.1. Порядок перечисления свойств

Спецификация ECMAScript не определяет порядок, в каком цикл `for/in` должен перечислять свойства объекта. Однако на практике реализации JavaScript во всех основных броузерах перечисляют свойства простых объектов в порядке, в каком они были определены, – когда ранее объявленные свойства перечисляются первыми. Если объект был создан с помощью литерала объекта, свойства перечисляются в том же порядке, в каком они следуют в литерале. В Интернете существуют сайты и библиотеки, которые опираются на такой порядок перечисления, поэтому маловероятно, что производители броузеров изменят его.

В абзаце выше описывается порядок перечисления свойств «простых» объектов. Однако в разных реализациях порядок перечисления может отличаться, если:

- объект наследует перечислимые свойства;
- объект имеет свойства, которые являются целочисленными индексами массива;
- использовалась инструкция `delete` для удаления существующих свойств объекта или
- использовался метод `Object.defineProperty()` (раздел 6.7) или аналогичный ему для изменения атрибутов свойства объекта.

Обычно (но не во всех реализациях) унаследованные свойства (раздел 6.2.2) перечисляются после всех неунаследованных, «собственных» свойств объекта, но они также перечисляются в порядке их определения. Если объект наследует свойства более чем от одного «прототипа» (раздел 6.1.3) – например, когда в его «цепочке прототипов» имеется более одного объекта, – свойства каждого объекта-прототипа в цепочке перечисляются в порядке их создания перед перечислением свойств следующего объекта. Некоторые (но не все) реализации перечисляют свойства массива в порядке возрастания чисел, а не в порядке их создания, но при наличии в массиве свойств с нечисловыми именами происходит возврат к перечислению в порядке создания свойств, то же самое происходит и в случае разреженных массивов (т. е. когда в массиве отсутствуют некоторые элементы).

5.6. Переходы

Еще одной категорией инструкций языка JavaScript являются инструкции перехода. Как следует из названия, эти инструкции заставляют интерпретатор JavaScript переходить в другое место в программном коде. Инструкция `break` заставляет интерпретатор перейти в конец цикла или другой инструкции. Инструкция `continue` заставляет интерпретатор пропустить оставшуюся часть тела цикла, перейти обратно в начало цикла и приступить к выполнению новой итерации. В языке JavaScript имеется возможность *помечать* инструкции именами, благодаря чему в инструкциях `break` и `continue` можно явно указывать, к какому циклу или к какой другой инструкции они относятся.

Инструкция `return` заставляет интерпретатор перейти из вызванной функции обратно в точку ее вызова и вернуть значение вызова. Инструкция `throw` возбуждает исключение и предназначена для работы в сочетании с инструкцией `try/catch/finally`, которая определяет блок программного кода для обработки исключения. Это достаточно сложная разновидность инструкции перехода: при появлении исключения интерпретатор переходит к ближайшему объемлющему обработчику исключений, который может находиться в той же функции или выше, в стеке возвратов вызванной функции.

Подробнее все эти инструкции перехода описываются в следующих подразделах.

5.6.1. Метки инструкций

Любая инструкция может быть *помечена* указанным перед ней идентификатором и двоеточием:

```
идентификатор: инструкция
```

Помечая инструкцию, вы тем самым даете ей имя, которое затем можно будет использовать в качестве ссылки в любом месте в программе. Пометить можно любую инструкцию, однако помечать имеет смысл только инструкции, имеющие тело, такие как циклы и условные инструкции. Присвоив имя циклу, его затем можно использовать в инструкциях `break` и `continue`, внутри цикла для выхода из него или для перехода в начало цикла, к следующей итерации. Инструкции `break` и `continue` являются единственными инструкциями в языке JavaScript, в которых можно указывать метки – о них подробнее рассказывается далее в этой главе. Ниже приводится пример инструкции `while` с меткой и инструкции `continue`, использующей эту метку:

```
mainloop: while(token != null) {
    // Программный код опущен...
    continue mainloop; // Переход к следующей итерации именованного цикла
    // Программный код опущен...
}
```

Идентификатор, используемый в качестве метки инструкции, может быть любым допустимым идентификатором JavaScript, кроме зарезервированного слова. Имена меток отделены от имен переменных и функций, поэтому в качестве меток допускается использовать идентификаторы, совпадающие с именами переменных или функций. Метки инструкций определены только внутри инструкций, к которым они применяются (и, конечно же, внутри вложенных в них инструкций). Вложенные инструкции не могут помечаться теми же идентификаторами, что и вмещающие их инструкции, но две независимые инструкции могут помечаться одинаковыми метками. Помеченные инструкции могут помечаться повторно. То есть любая инструкция может иметь множество меток.

5.6.2. Инструкция break

Инструкция `break` приводит к немедленному выходу из самого внутреннего цикла или инструкции `switch`. Синтаксис ее прост:

```
break;
```

Поскольку инструкция `break` приводит к выходу из цикла или инструкции `switch`, такая форма `break` допустима только внутри этих инструкций.

Выше мы уже видели примеры использования инструкции `break` внутри инструкции `switch`. В циклах она обычно используется для немедленного выхода из цикла, когда по каким-либо причинам требуется завершить выполнение цикла. Когда цикл имеет очень сложное условие завершения, зачастую проще бывает реализовать эти условия с помощью инструкций `break`, чем пытаться выразить их в одном условном выражении цикла. Следующий пример пытается отыскать элемент массива с определенным значением. Цикл завершается обычным образом по достижении конца массива или с помощью инструкции `break`, как только будет найдено искомое значение:

```
for(var i = 0; i < a.length; i++) {
    if (a[i] == target) break;
}
```

В языке JavaScript допускается указывать имя метки за ключевым словом `break` (идентификатор без двоеточия):

```
break имя_метки;
```

Когда инструкция break используется с меткой, она выполняет переход в конец именованной инструкции или прекращение ее выполнения. В случае отсутствия инструкции с указанной меткой попытка использовать такую форму инструкции break порождает синтаксическую ошибку. Именованная инструкция не обязана быть циклом или инструкцией switch: инструкция break с меткой может выполнять «выход» из любой вмещающей ее инструкции. Объемлющая инструкция может даже быть простым блоком инструкций, заключенным в фигурные скобки исключительно с целью пометить его.

Между ключевым словом break и именем метки не допускается вставлять символ перевода строки. Дело в том, что интерпретатор JavaScript автоматически вставляет пропущенные точки с запятой: если разбить строку программного кода между ключевым словом break и следующей за ним меткой, интерпретатор предположит, что имелась в виду простая форма этой инструкции без метки, и добавит точку с запятой (раздел 2.5).

Инструкция break с меткой необходима, только когда требуется прервать выполнение инструкции, не являющейся ближайшим объемлющим циклом или инструкцией switch. Следующий фрагмент демонстрирует это:

```
var matrix = getData(); // Получить 2-мерный массив чисел откуда-нибудь
// Найти сумму всех чисел в матрице.
var sum = 0, success = false;
// Пометить инструкцию, выполнение которой требуется прервать в случае ошибки
compute_sum: if (matrix) {
    for(var x = 0; x < matrix.length; x++) {
        var row = matrix[x];
        if (!row) break compute_sum;
        for(var y = 0; y < row.length; y++) {
            var cell = row[y];
            if (isNaN(cell)) break compute_sum;
            sum += cell;
        }
    }
    success = true;
}
// Здесь инструкция break выполняет переход. Если будет выполнено условие
// success == false, значит, что-то не так в полученной матрице.
// В противном случае переменная sum будет содержать сумму всех элементов матрицы.
```

Наконец, обратите внимание, что инструкция break, с меткой или без нее, не может передавать управление через границы функций. Например, нельзя пометить инструкцию объявления функции и затем использовать эту метку внутри функции.

5.6.3. Инструкция continue

Инструкция continue схожа с инструкцией break. Однако вместо выхода из цикла инструкция continue запускает новую итерацию цикла. Синтаксис инструкции continue столь же прост, как и синтаксис инструкции break:

```
continue;
```

Инструкция continue может также использоваться с меткой:

```
continue имя_метки;
```

Инструкция continue, как в форме без метки, так и с меткой, может использоваться только в теле цикла. Использование ее в любых других местах приводит к синтаксической ошибке.

Когда выполняется инструкция continue, текущая итерация цикла прерывается и начинается следующая. Для разных типов циклов это означает разное:

- В цикле while указанное в начале цикла *выражение* проверяется снова, и если оно равно true, тело цикла выполняется с начала.
- В цикле do/while происходит переход в конец цикла, где перед повторным выполнением цикла снова проверяется условие.
- В цикле for вычисляется выражение *инкремента* и снова вычисляется выражение *проверки*, чтобы определить, следует ли выполнять следующую итерацию.
- В цикле for/in цикл начинается заново с присвоением указанной переменной имени следующего свойства.

Обратите внимание на различия в поведении инструкции continue в циклах while и for: цикл while возвращается непосредственно к своему условию, а цикл for сначала вычисляет выражение *инкремента*, а затем возвращается к условию. Ранее при обсуждении цикла for объяснялось поведение цикла for в терминах «эквивалентного» цикла while. Поскольку инструкция continue ведет себя в этих двух циклах по-разному, точно имитировать цикл for с помощью одного цикла while невозможно.

В следующем примере показано использование инструкции continue без метки для выхода из текущей итерации цикла в случае ошибки:

```
for(i = 0; i < data.length; i++) {
    if (!data[i]) continue;        // Не обрабатывать неопределенные данные
    total += data[i];
}
```

Инструкция continue, как и break, может применяться во вложенных циклах в форме, включающей метку, и тогда заново запускаемым циклом необязательно будет цикл, непосредственно содержащий инструкцию continue. Кроме того, как и для инструкции break, переводы строк между ключевым словом continue и именем метки не допускаются.

5.6.4. Инструкция return

Как вы помните, вызов функции является выражением и подобно всем выражениям имеет значение. Инструкция return внутри функций служит для определения значения, возвращаемого функцией. Инструкция return имеет следующий синтаксис:

```
return выражение;
```

Инструкция return может располагаться только в теле функции. Присутствие ее в любом другом месте является синтаксической ошибкой. Когда выполняется инструкция return, функция возвращает значение *выражения* вызывающей программе. Например:

```
function square(x) { return x*x; }  // Функция с инструкцией return
square(2)                           // Этот вызов вернет 4
```

Если функция не имеет инструкции `return`, при ее вызове интерпретатор будет выполнять инструкции в теле функции одну за другой, пока не достигнет конца функции, и затем вернет управление вызвавшей ее программе. В этом случае выражение вызова вернет значение `undefined`. Инструкция `return` часто является последней инструкцией в функции, но это совершенно необязательно: функция вернет управление вызывающей программе, как только будет достигнута инструкция `return`, даже если за ней следуют другие инструкции в теле функции.

Инструкция `return` может также использоваться без *выражения*, тогда она просто прерывает выполнение функции и возвращает значение `undefined` вызывающей программе. Например:

```
function display_object(o) {
    // Сразу же вернуть управление, если аргумент имеет значение null или undefined
    if (!o) return;
    // Здесь находится оставшаяся часть функции...
}
```

Из-за того что интерпретатор JavaScript автоматически вставляет точки с запятой, нельзя разделять переводом строки инструкцию `return` и следующее за ней выражение.

5.6.5. Инструкция throw

Исключение – это сигнал, указывающий на возникновение какой-либо исключительной ситуации или ошибки. *Возбуждение* исключения (throw) – это способ просигнализировать о такой ошибке или исключительной ситуации. *Перехватить* исключение (catch) – значит обработать его, т. е. предпринять действия, необходимые или подходящие для восстановления после исключения. В JavaScript исключения возбуждаются в тех случаях, когда возникает ошибка времени выполнения и когда программа явно возбуждает его с помощью инструкции `throw`. Исключения перехватываются с помощью инструкции `try/catch/finally`, которая описана в следующем разделе.

Инструкция `throw` имеет следующий синтаксис:

```
throw выражение;
```

Результатом *выражения* может быть значение любого типа. Инструкции `throw` можно передать число, представляющее код ошибки, или строку, содержащую текст сообщения об ошибке. Интерпретатор JavaScript возбуждает исключения, используя экземпляр класса `Error` одного из его подклассов, и вы также можете использовать подобный подход. Объект `Error` имеет свойство `name`, определяющее тип ошибки, и свойство `message`, содержащее строку, переданную функции-конструктору (смотрите описание класса `Error` в справочном разделе). Ниже приводится пример функции, которая возбуждает объект `Error` при вызове с недопустимым аргументом:

```
function factorial(x) {
    // Если входной аргумент не является допустимым значением, возбуждается исключение!
    if (x < 0) throw new Error("x не может быть отрицательным");
    // В противном случае значение вычисляется и возвращается нормальным образом
    for(var f = 1; x > 1; f *= x, x--) /* пустое тело цикла */ ;
    return f;
}
```

Когда возбуждается исключение, интерпретатор JavaScript немедленно прерывает нормальное выполнение программы и переходит к ближайшему[1] обработчику исключений. В обработчиках исключений используется конструкция `catch` инструкции `try/catch/finally`, описание которой приведено в следующем разделе. Если блок программного кода, в котором возникло исключение, не имеет соответствующей конструкции `catch`, интерпретатор анализирует следующий внешний блок программного кода и проверяет, связан ли с ним обработчик исключений. Это продолжается до тех пор, пока обработчик не будет найден. Если исключение генерируется в функции, не содержащей инструкции `try/catch/finally`, предназначенной для его обработки, то исключение распространяется выше, в программный код, вызвавший функцию. Таким образом исключения распространяются по лексической структуре методов JavaScript вверх по стеку вызовов. Если обработчик исключения так и не будет найден, исключение рассматривается как ошибка и о ней сообщается пользователю.

5.6.6. Инструкция try/catch/finally

Инструкция `try/catch/finally` реализует механизм обработки исключений в JavaScript. Конструкция `try` в этой инструкции просто определяет блок кода, в котором обрабатываются исключения. За блоком `try` следует конструкция `catch` с блоком инструкций, вызываемых, если где-либо в блоке `try` возникает исключение. За конструкцией `catch` следует блок `finally`, содержащий программный код, выполняющий заключительные операции, который гарантированно выполняется независимо от того, что происходит в блоке `try`. И блок `catch`, и блок `finally` не являются обязательными, однако после блока `try` должен обязательно присутствовать хотя бы один из них. Блоки `try`, `catch` и `finally` начинаются и заканчиваются фигурными скобками. Это обязательная часть синтаксиса, и она не может быть опущена, даже если между ними содержится только одна инструкция.

Следующий фрагмент иллюстрирует синтаксис и назначение инструкции `try/catch/finally`:

```
try {
    // Обычно этот код без сбоев работает от начала до конца.
    // Но в какой-то момент в нем может быть сгенерировано исключение
    // либо непосредственно с помощью инструкции throw, либо косвенно -
    // вызовом метода, генерирующего исключение.
}
catch (e) {
    // Инструкции в этом блоке выполняются тогда и только тогда, когда в блоке try
    // возникает исключение. Эти инструкции могут использовать локальную переменную e,
    // ссылающуюся на объект Error или на другое значение, указанное в инструкции throw.
    // Этот блок может либо некоторым образом обработать исключение, либо
    // проигнорировать его, делая что-то другое, либо заново сгенерировать
    // исключение с помощью инструкции throw.
}
finally {
    // Этот блок содержит инструкции, которые выполняются всегда, независимо от того,
    // что произошло в блоке try. Они выполняются, если блок try завершился:
```

---

[1] **К самому внутреннему по вложенности охватывающему обработчику исключений. –** ***Прим. науч. ред.***

```
    //     1) как обычно, достигнув конца блока
    //     2) из-за инструкции break, continue или return
    //     3) с исключением, обработанным приведенным в блоке catch выше
    //     4) с неперехваченным исключением, которое продолжает свое
    //        распространение на более высокие уровни
}
```

Обратите внимание, что за ключевым словом `catch` следует идентификатор в скобках. Этот идентификатор похож на параметр функции. Когда будет перехвачено исключение, этому параметру будет присвоено исключение (например, объект `Error`). В отличие от обычной переменной идентификатор, ассоциированный с конструкцией `catch`, существует только в теле блока `catch`.

Далее приводится более реалистичный пример инструкции `try/catch`. В нем вызываются метод `factorial()`, определенный в предыдущем разделе, и методы `prompt()` и `alert()` клиентского JavaScript для организации ввода и вывода:

```
try {
    // Запросить число у пользователя
    var n = Number(prompt("Введите положительное число", ""));
    // Вычислить факториал числа, предполагая, что входные данные корректны
    var f = factorial(n);
    // Вывести результат
    alert(n + "! = " + f);
}
catch (ex) {   // Если данные некорректны, управление будет передано сюда
    alert(ex); // Сообщить пользователю об ошибке
}
```

Это пример инструкции `try/catch` без конструкции `finally`. Хотя `finally` используется не так часто, как `catch`, тем не менее иногда эта конструкция оказывается полезной. Однако ее поведение требует дополнительных объяснений. Блок `finally` гарантированно исполняется, если исполнялась хотя бы какая-то часть блока `try`, независимо от того, каким образом завершилось выполнение программного кода в блоке `try`. Эта возможность обычно используется для выполнения заключительных операций после выполнения программного кода в предложении `try`.

В обычной ситуации управление доходит до конца блока `try`, а затем переходит к блоку `finally`, который выполняет необходимые заключительные операции. Если управление вышло из блока `try` как результат выполнения инструкций `return`, `continue` или `break`, перед передачей управления в другое место выполняется блок `finally`.

Если в блоке `try` возникает исключение и имеется соответствующий блок `catch` для его обработки, управление сначала передается в блок `catch`, а затем – в блок `finally`. Если отсутствует локальный блок `catch`, то управление сначала передается в блок `finally`, а затем переходит на ближайший внешний блок `catch`, который может обработать исключение.

Если сам блок `finally` передает управление с помощью инструкции `return`, `continue`, `break` или `throw` или путем вызова метода, генерирующего исключение, незаконченная команда на передачу управления отменяется и выполняется новая. Например, если блок `finally` сгенерирует исключение, это исключение заменит любое ранее сгенерированное исключение. Если в блоке `finally` имеется инструкция `return`, произойдет нормальный выход из метода, даже если генерировалось исключение, которое не было обработано.

Конструкции `try` и `finally` могут использоваться вместе без конструкции `catch`. В этом случае блок `finally` – это просто набор инструкций, выполняющих заключительные операции, который будет гарантированно выполнен независимо от наличия в блоке `try` инструкции `break`, `continue` или `return`. Напомню, из-за различий в работе инструкции `continue` в разных циклах невозможно написать цикл `while`, полностью имитирующий работу цикла `for`. Однако если добавить инструкцию `try/finally`, можно написать цикл `while`, который будет действовать точно так же, как цикл `for`, и корректно обрабатывать инструкцию `continue`:

```
// Имитация цикла for( инициализация ; проверка ; инкремент ) тело цикла;
инициализация ;
while( проверка ) {
    try { тело цикла ; }
    finally { инкремент ; }
}
```

Обратите однако внимание, что тело цикла `while`, содержащее инструкцию `break`, будет вести себя несколько иначе (из-за выполнения лишней операции инкремента перед выходом), чем тело цикла `for`, поэтому даже используя конструкцию `finally`, невозможно точно сымитировать цикл `for` с помощью цикла `while`.

5.7. Прочие инструкции

В этом разделе описываются три остальные инструкции языка JavaScript – `with`, `debugger` и `use strict`.

5.7.1. Инструкция with

В разделе 3.10.3 мы обсуждали область видимости переменных и цепочки областей видимости – список объектов, в которых выполняется поиск при разрешении имен переменных. Инструкция `with` используется для временного изменения цепочки областей видимости. Она имеет следующий синтаксис:

```
with (обьект)
    инструкция
```

Эта инструкция добавляет *обьект* в начало цепочки областей видимости, выполняет *инструкцию*, а затем восстанавливает первоначальное состояние цепочки.

Инструкция `with` не может использоваться в строгом режиме (раздел 5.7.3) и не рекомендуется к использованию в нестрогом режиме: избегайте ее использования по мере возможности. Программный код JavaScript, в котором используется инструкция `with`, сложнее поддается оптимизации и наверняка будет работать медленнее, чем эквивалентный программный код без инструкции `with`.

На практике инструкция `with` упрощает работу с глубоко вложенными иерархиями объектов. В клиентском JavaScript вам наверняка придется вводить выражения, как показано ниже, чтобы обратиться к элементам HTML-формы:

```
document.forms[0].address.value
```

Если подобные выражения потребуется записать много раз, можно воспользоваться инструкцией `with`, чтобы добавить объект формы в цепочку областей видимости:

```
with(document.forms[0]) {
    // Далее следуют обращения к элементам формы непосредственно, например:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

Этот прием сокращает объем текста программы – больше не надо указывать фрагмент `document.forms[0]` перед каждым именем свойства. Этот объект представляет собой временную часть цепочки областей видимости и автоматически участвует в поиске, когда JavaScript требуется разрешить идентификаторы, такие как `address`. Избежать применения инструкции `with` достаточно просто, если записать предыдущий пример, как показано ниже:

```
var f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Имейте в виду, что цепочка областей видимости используется только для поиска идентификаторов и не используется при их создании. Взгляните на следующий пример:

```
with(o) x = 1;
```

Если объект `o` имеет свойство `x`, то данный программный код присвоит значение `1` этому свойству. Но если `x` не является свойством объекта `o`, данный программный код выполнит то же действие, что и инструкция `x = 1` без инструкции `with`. Он присвоит значение локальной или глобальной переменной с именем `x` или создаст новое свойство глобального объекта. Инструкция `with` обеспечивает более короткую форму записи операций чтения свойств объекта `o`, но не создания новых свойств этого объекта.

5.7.2. Инструкция debugger

Инструкция `debugger` обычно ничего не делает. Однако если имеется и запущена программа-отладчик, реализация JavaScript может (но не обязана) выполнять некоторые отладочные операции. Обычно эта инструкция действует подобно точке останова: интерпретатор JavaScript приостанавливает выполнение программного кода, и вы можете с помощью отладчика вывести значения переменных, ознакомиться с содержимым стека вызовов и т. д. Допустим, к примеру, что ваша функция `f()` порождает исключение, потому что она вызывается с неопределенным аргументом, а вы никак не можете определить, из какой точки программы производится этот вызов. Чтобы решить эту проблему, можно было бы изменить определение функции `f()`, чтобы она начиналась строкой, как показано ниже:

```
function f(o) {
    if (o === undefined) debugger; // Временная строка для отладки
                                   // Далее продолжается тело функции.
}
```

Теперь, когда `f()` будет вызвана без аргумента, ее выполнение будет приостановлено, и вы сможете воспользоваться отладчиком и просмотреть стек вызовов, чтобы отыскать место, откуда был выполнен некорректный вызов.

Официально инструкция `debugger` была добавлена в язык стандартом ECMAScript 5, но производители основных броузеров реализовали ее уже достаточно давно. Обратите внимание, что недостаточно иметь отладчик: инструкция `debugger` не запускает отладчик автоматически. Однако, если отладчик уже запущен, эта инструкция будет действовать как точка останова. Если, к примеру, воспользоваться расширением Firebug для Firefox, это расширение должно быть активировано для веб-страницы, которую требуется отладить, и только в этом случае инструкция `debugger` будет работать.

5.7.3. "use strict"

`"use strict"` – это *директива*, введенная стандартом ECMAScript 5. Директивы не являются инструкциями (но достаточно близки, чтобы включить описание `"use strict"` в эту главу). Между обычными инструкциями и директивой `"use strict"` существует два важных отличия:

- Она не включает никаких зарезервированных слов языка: директива – это лишь выражение, содержащее специальный строковый литерал (в одиночных или двойных кавычках). Интерпретаторы JavaScript, не соответствующие стандарту ECMAScript 5, будут интерпретировать ее как простое выражение без побочных эффектов и ничего не будут делать. В будущих версиях стандарта ECMAScript, как ожидается, слово `use` будет переведено в разряд ключевых слов, что позволит опустить кавычки.
- Она может появляться только в начале сценария или в начале тела функции, перед любыми другими инструкциями. Однако она не обязательно должна находиться в самой первой строке сценария или функции: директиве `"use strict"` могут предшествовать или следовать за ней другие строковые выражения-литералы, а различные реализации JavaScript могут интерпретировать эти строковые литералы как директивы, определяемые этими реализациями. Строковые литералы, следующие за первой обычной инструкцией в сценарии или функции, интерпретируются как обычные выражения – они могут не восприниматься как директивы и не оказывать никакого эффекта.

Назначение директивы `"use strict"` состоит в том, чтобы показать, что следующий за ней программный код (в сценарии или функции) является *строгим кодом*. Строгим считается программный код верхнего уровня (не внутри функций), если в сценарии имеется директива `"use strict"`. Строгим считается тело функции, если она определяется внутри строгого программного кода или если она содержит директиву `"use strict"`. Строгим считается программный код, передаваемый методу `eval()`, если вызов `eval()` выполняется из строгого программного кода или если строка с кодом содержит директиву `"use strict"`.

Строгий программный код выполняется в *строгом режиме*. Согласно стандарту ECMAScript 5, строгий режим определяет ограниченное подмножество языка, благодаря чему исправляет некоторые недостатки языка, а также обеспечивает более строгую проверку на наличие ошибок и повышенный уровень безопасности. Ниже перечислены различия между строгим и нестрогим режимами (первые три имеют особенно большое значение):

- **В строгом режиме не допускается использование инструкции** `with`**.**
- **В строгом режиме все переменные должны объявляться: если попытаться присвоить значение идентификатору, который не является объявленной переменной, функцией, параметром функции, параметром конструкции** `catch` **или свойством глобального объекта, возбуждается исключение** `ReferenceError`**. (В нестрогом режиме такая попытка просто создаст новую глобальную переменную и добавит ее в виде свойства в глобальный объект.)**
- **В строгом режиме функции, которые вызываются как функции (а не как методы), получают в ссылке** `this` **значение** `undefined`**. (В нестрогом режиме функции, которые вызываются как функции, всегда получают в ссылке** `this` **глобальный объект.) Это отличие можно использовать, чтобы определить, поддерживает ли та или иная реализация строгий режим:**

  ```
  var hasStrictMode = (function() { "use strict"; return this===undefined}());
  ```

  Кроме того, когда функция вызывается в строгом режиме с помощью `call()` **или** `apply()`**, значение ссылки** `this` **в точности соответствует значению, переданному в первом аргументе функции** `call()` **или** `apply()`**. (В нестрогом режиме значения** `null` **и** `undefined` **замещаются ссылкой на глобальный объект, а простые значения преобразуются в объекты.)**
- **В строгом режиме попытки присвоить значения свойствам, недоступным для записи, или создать новые свойства в нерасширяемых объектах порождают исключение** `TypeError`**. (В нестрогом режиме эти попытки просто игнорируются.)**
- **В строгом режиме программный код, передаваемый функции** `eval()`**, не может объявлять переменные или функции в области видимости вызывающего программного кода, как это возможно в нестрогом режиме. Вместо этого переменные и функции помещаются в новую область видимости, создаваемую для функции** `eval()`**. Эта область видимости исчезает, как только** `eval()` **вернет управление.**
- **В строгом режиме объект** `arguments` **(раздел 8.3.2) в функции хранит статическую копию значений, переданных функции. В нестрогом режиме объект** `arguments` **ведет себя иначе – элементы массива** `arguments` **и именованные параметры функции ссылаются на одни и те же значения.**
- **В строгом режиме возбуждается исключение** `SyntaxError`**, если оператору** `delete` **передать неквалифицированный идентификатор, такой как имя переменной, функции или параметра функции. (В нестрогом режиме такое выражение** `delete` **не выполнит никаких действий и вернет** `false`**.)**
- **В строгом режиме попытка удалить ненастраиваемое свойство приведет к исключению** `TypeError`**. (В нестрогом режиме эта попытка просто завершится неудачей и выражение** `delete` **вернет** `false`**.)**
- **В строгом режиме попытка определить в литерале объекта два или более свойств с одинаковыми именами считается синтаксической ошибкой. (В нестрогом режиме ошибка не возникает.)**
- **В строгом режиме определение двух или более параметров с одинаковыми именами в объявлении функции считается синтаксической ошибкой. (В нестрогом режиме ошибка не возникает.)**

- В строгом режиме не допускается использовать литералы восьмеричных целых чисел (начинающихся с 0, за которым не следует символ `x`). (В нестрогом режиме некоторые реализации позволяют использовать восьмеричные литералы.)
- В строгом режиме идентификаторы `eval` и `arguments` интерпретируются как ключевые слова, и вы не можете изменить их значения. Вы сможете присвоить значения этим идентификаторам, объявив их как переменные, использовав их в качестве имен функций, имен параметров функций или идентификаторов блока `catch`.
- В строгом режиме ограничивается возможность просмотра стека вызовов. Попытки обратиться к свойствам `arguments.caller` и `arguments.callee` в строгом режиме возбуждают исключение `TypeError`. Попытки прочитать свойства `caller` и `arguments` функций в строгом режиме также возбуждают исключение `TypeError`. (Некоторые реализации определяют эти свойства в нестрогих функциях.)

5.8. Итоговая таблица JavaScript-инструкций

В этой главе были представлены все инструкции языка JavaScript. В табл. 5.1 содержится перечень этих инструкций с указанием синтаксиса и назначения каждой из них.

Таблица 5.1. Синтаксис инструкций JavaScript

Инструкция	Синтаксис	Назначение
break	`break` [*имя_метки*];	Выход из самого внутреннего цикла, инструкции `switch` или инструкции с именем *имя_метки*
case	`case` *выражение*:	Метка для инструкции внутри конструкции `switch`
continue	`continue` [*имя_метки*];	Переход к следующей итерации самого внутреннего цикла или цикла, помеченного меткой *имя_метки*
debugger	`debugger;`	Точка останова отладчика
default	`default:`	Метка инструкции по умолчанию, внутри инструкции `switch`
do/while	`do` *инструкция* `while` (*выражение*);	Альтернатива циклу `while`
пустая инструкция	;	Ничего не делает
for	`for` (*инициализация*; *проверка*; *инкремент*) *инструкция*	Простой в использовании цикл
for/in	`for` (*переменная* `in` *обьект*) *инструкция*	Цикл по свойствам объекта

function	`function` *имя_функции*([парам[,...]])
	{
	тело
	}
		Объявление функции с именем *имя_функции*

Таблица 5.1 (продолжение)

Инструкция	Синтаксис	Назначение
if/else	`if (`*выражение*`)` *инструкция1* `[else` *инструкция2*`]`	Выполняет *инструкцию1* или *инструкцию2*
метка	*идентификатор*: *инструкция*	Дает *инструкции* имя *идентификатор*
return	`return [`*выражение*`];`	Возвращает значение из функции
switch	`switch (`*выражение*`) {` *инструкции* `}`	Многопозиционное ветвление для инструкций с метками `case` и `default`
throw	`throw` *выражение*;	Генерирует исключения

try	`try{` *инструкции* `}`
	`[catch{` *обработчик исключений* `}]`
	`[finally {` *заключит. операции* `}]`
		Обработка исключений

use strict	`"use strict"`	Применение строгого режима, накладывающего ограничения на сценарии или функции
var	`var` *имя* `[ =` *выражение* `] [ ,... ];`	Объявление и инициализация одной или более переменных
while	`while (`*выражение*`)` *инструкция*	Базовая конструкция цикла
with	`with (`*объект*`)` *инструкция*	Расширение цепочки областей видимости. (Не рекомендуется к использованию.)

6

Объекты

Объект является фундаментальным типом данных в языке JavaScript. Объект – это составное значение: он объединяет в себе набор значений (простых значений или других объектов) и позволяет сохранять и извлекать эти значения по именам. Объект является неупорядоченной коллекцией *свойств*, каждое из которых имеет имя и значение. Имена свойств являются строками, поэтому можно сказать, что объекты отображают строки в значения. Такое отображение строк в значения может называться по-разному: возможно, вы уже знакомы с такой фундаментальной структурой данных, как «хеш», «словарь» или «ассоциативный массив». Однако объект представляет собой нечто большее, чем простое отображение строк в значения. Помимо собственных свойств объекты в языке JavaScript могут также наследовать свойства от других объектов, известных под названием «прототипы». Методы объекта – это типичные представители унаследованных свойств, а «наследование через прототипы» является ключевой особенностью языка JavaScript.

Объекты в языке JavaScript являются динамическими – обычно они позволяют добавлять и удалять свойства – но они могут использоваться также для имитации статических объектов и «структур», которые имеются в языках программирования со статической системой типов. Кроме того, они могут использоваться (если не учитывать, что объекты отображают строки в значения) для представления множеств строк.

Любое значение в языке JavaScript, не являющееся строкой, числом, `true`, `false`, `null` или `undefined`, является объектом. И даже строки, числа и логические значения, не являющиеся объектами, могут вести себя как неизменяемые объекты (раздел 3.6).

Как вы помните, в разделе 3.7 говорилось, что объекты являются изменяемыми значениями и операции с ними выполняются по ссылке, а не по значению. Если переменная `x` ссылается на объект, и выполняется инструкция `var y = x;`, в переменную `y` будет записана ссылка на тот же самый объект, а не его копия. Любые изменения, выполняемые в объекте с помощью переменной `y`, будут также отражаться на переменной `x`.

Наиболее типичными операциями с объектами являются создание объектов, назначение, получение, удаление, проверка и перечисление их свойств. Эти базовые операции описываются в начальных разделах этой главы. В следующих за ними разделах будут рассматриваться более сложные темы, многие из которых имеют прямое отношение к стандарту ECMAScript 5.

Свойство имеет имя и значение. Именем свойства может быть любая строка, включая и пустую строку, но объект не может иметь два свойства с одинаковыми именами. Значением свойства может быть любое значение, допустимое в языке JavaScript, или (в ECMAScript 5) функция чтения или записи (или обе). Поближе с функциями чтения и записи свойств мы познакомимся в разделе 6.6. В дополнение к именам и значениям каждое свойство имеет ряд ассоциированных с ним значений, которые называют *атрибутами свойства*:

- Атрибут `writable` определяет доступность значения свойства для записи.
- Атрибут `enumerable` определяет доступность имени свойства для перечисления в цикле `for/in`.
- Атрибут `configurable` определяет возможность настройки, т. е. удаления свойства и изменения его атрибутов.

До появления стандарта ECMAScript 5 все свойства в объектах, создаваемые программой, доступны для записи, перечисления и настройки. В ECMAScript 5 предусматривается возможность настройки атрибутов ваших свойств. Как это делается, описывается в разделе 6.7.

В дополнение к свойствам каждый объект имеет три *атрибута объекта*:

- Атрибут `prototype` содержит ссылку на другой объект, от которого наследуются свойства.
- Атрибут `class` содержит строку с именем класса объекта и определяет тип объекта.
- Флаг `extensible` (в ECMAScript 5) указывает на возможность добавления новых свойств в объект.

Поближе с прототипами и механизмом наследования свойств мы познакомимся в разделах 6.1.3 и 6.2.2, а более детальное обсуждение всех трех атрибутов объектов вы найдете в разделе 6.8.

Наконец, ниже приводится описание некоторых терминов, которые помогут нам различать три обширные категории объектов в языке JavaScript и два типа свойств:

- *Объект базового языка* – это объект или класс объектов, определяемый спецификацией ECMAScript. Массивы, функции, даты и регулярные выражения (например) являются объектами базового языка.
- *Объект среды выполнения* – это объект, определяемый средой выполнения (такой как веб-броузер), куда встроен интерпретатор JavaScript. Объекты `HTMLElement`, представляющие структуру веб-страницы в клиентском JavaScript, являются объектами среды выполнения. Объекты среды выполнения могут также быть объектами базового языка, например, когда среда выполнения определяет методы, которые являются обычными объектами `Function` базового языка JavaScript.
- *Пользовательский объект* – любой объект, созданный в результате выполнения программного кода JavaScript.

- *Собственное свойство* – это свойство, определяемое непосредственно в данном объекте.
- *Унаследованное свойство* – это свойство, определяемое прототипом объекта.

6.1. Создание объектов

Объекты можно создавать с помощью литералов объектов, ключевого слова `new` и (в ECMAScript 5) функции `Object.create()`. Все эти приемы описываются в следующих разделах.

6.1.1. Литералы объектов

Самый простой способ создать объект заключается во включении в программу литерала объекта. *Литерал объекта* – это заключенный в фигурные скобки список свойств (пар имя/значение), разделенных запятыми. Именем свойства может быть идентификатор или строковый литерал (допускается использовать пустую строку). Значением свойства может быть любое выражение, допустимое в JavaScript, – значение выражения (это может быть простое значение или объект) станет значением свойства. Ниже приводится несколько примеров создания объектов:

```
var empty = {};                          // Объект без свойств
var point = { x:0, y:0 };                // Два свойства
var point2 = { x:point.x, y:point.y+1 }; // Более сложные значения
var book = {
    "main title": "JavaScript",          // Имена свойств с пробелами
    'sub-title': "The Definitive Guide", // и дефисами, поэтому используются
                                         // строковые литералы
    "for": "all audiences",              // for - зарезервированное слово,
                                         // поэтому в кавычках
    author: {                            // Значением этого свойства является
        firstname: "David",              // объект. Обратите внимание, что
        surname: "Flanagan"              // имена этих свойств без кавычек.
    }
};
```

В ECMAScript 5 (и в некоторых реализациях ECMAScript 3) допускается использовать зарезервированные слова в качестве имен свойств без кавычек. Однако в целом имена свойств, совпадающие с зарезервированными словами, в ECMAScript 3 должны заключаться в кавычки. В ECMAScript 5 последняя запятая, следующая за последним свойством в литерале объекта, игнорируется. В большинстве реализаций ECMAScript 3 завершающие запятые также игнорируются, но IE интерпретирует их наличие как ошибку.

Литерал объекта – это выражение, которое создает и инициализирует новый объект всякий раз, когда производится вычисление этого выражения. Значение каждого свойства вычисляется заново, когда вычисляется значение литерала. Это означает, что с помощью единственного литерала объекта можно создать множество новых объектов, если этот литерал поместить в тело цикла или функции, которая будет вызываться многократно, и что значения свойств этих объектов могут отличаться друг от друга.

6.1.2. Создание объектов с помощью оператора new

Оператор `new` создает и инициализирует новый объект. За этим оператором должно следовать имя функции. Функция, используемая таким способом, называется *конструктором* и служит для инициализации вновь созданного объекта. Базовый JavaScript включает множество встроенных конструкторов для создания объектов базового языка. Например:

```
var o = new Object(); // Создать новый пустой объект: то же, что и {}.
var a = new Array();  // Создать пустой массив: то же, что и [].
var d = new Date();   // Создать объект Date, представляющий текущее время
var r = new RegExp("js"); // Создать объект RegExp для операций
                          // сопоставления с шаблоном.
```

Помимо этих встроенных конструкторов имеется возможность определять свои собственные функции-конструкторы для инициализации вновь создаваемых объектов. О том, как это делается, рассказывается в главе 9.

6.1.3. Прототипы

Прежде чем перейти к третьему способу создания объектов, необходимо сделать паузу, чтобы познакомиться с прототипами. Каждый объект в языке JavaScript имеет второй объект (или `null`, но значительно реже), ассоциированный с ним. Этот второй объект называется прототипом, и первый объект наследует от прототипа его свойства.

Все объекты, созданные с помощью литералов объектов, имеют один и тот же объект-прототип, на который в программе JavaScript можно сослаться так: `Object.prototype`. Объекты, созданные с помощью ключевого слова `new` и вызова конструктора, в качестве прототипа получают значение свойства `prototype` функции-конструктора. Поэтому объект, созданный выражением `new Object()`, наследует свойства объекта `Object.prototype`, как если бы он был создан с помощью литерала в фигурных скобках `{}`. Аналогично прототипом объекта, созданного выражением `new Array()`, является `Array.prototype`, а прототипом объекта, созданного выражением `new Date()`, является `Date.prototype`.

`Object.prototype` – один из немногих объектов, которые не имеют прототипа: у него нет унаследованных свойств. Другие объекты-прототипы являются самыми обычными объектами, имеющими собственные прототипы. Все встроенные конструкторы (и большинство пользовательских конструкторов) наследуют прототип `Object.prototype`. Например, `Date.prototype` наследует свойства от `Object.prototype`, поэтому объект `Date`, созданный выражением `new Date()`, наследует свойства от обоих прототипов, `Date.prototype` и `Object.prototype`. Такая связанная последовательность объектов-прототипов называется *цепочкой прототипов*.

Описание механизма наследования свойств приводится в разделе 6.2.2. Как получить ссылку на прототип объекта, рассказывается в разделе 6.8.1. А в главе 9 более детально будет обсуждаться связь между прототипами и конструкторами: там будет показано, как определять новые «классы» объектов посредством объявления функций-конструкторов и как записывать ссылку на объект-прототип в их свойство `prototype` для последующего использования «экземплярами», созданными с помощью этого конструктора.

6.1.4. Object.create()

Стандарт ECMAScript 5 определяет метод `Object.create()`, который создает новый объект и использует свой первый аргумент в качестве прототипа этого объекта. Дополнительно `Object.create()` может принимать второй необязательный аргумент, описывающий свойства нового объекта. Описание этого второго аргумента приводится в разделе 6.7.

`Object.create()` является статической функцией, а не методом, вызываемым относительно некоторого конкретного объекта. Чтобы вызвать эту функцию, достаточно передать ей желаемый объект-прототип:

```
var o1 = Object.create({x:1, y:2});      // o1 наследует свойства x и y.
```

Чтобы создать объект, не имеющий прототипа, можно передать значение `null`, но в этом случае вновь созданный объект не унаследует ни каких-либо свойств, ни базовых методов, таких как `toString()` (а это означает, что этот объект нельзя будет использовать в выражениях с оператором +):

```
var o2 = Object.create(null);            // o2 не наследует ни свойств, ни методов.
```

Если в программе потребуется создать обычный пустой объект (который, например, возвращается литералом {} или выражением `new Object()`), передайте в первом аргументе `Object.prototype`:

```
var o3 = Object.create(Object.prototype); // o3 подобен объекту, созданному
                                          // с помощью {} или new Object().
```

Возможность создавать новые объекты с произвольными прототипами (скажем иначе: возможность создавать «наследников» от любых объектов) является мощным инструментом, действие которого можно имитировать в ECMAScript 3 с помощью функции, представленной в примере 6.1.[1]

Пример 6.1. Создание нового объекта, наследующего прототип

```
// inherit() возвращает вновь созданный объект, наследующий свойства
// объекта-прототипа p. Использует функцию Object.create() из ECMAScript 5,
// если она определена, иначе используется более старый прием.
function inherit(p) {
    if (p == null) throw TypeError(); // p не может быть значением null
    if (Object.create)                // Если Object.create() определена...
        return Object.create(p);      // использовать ее.
    var t = typeof p;                 // Иначе выяснить тип и проверить его
    if (t !== "object" && t !== "function") throw TypeError();
    function f() {};                  // Определить фиктивный конструктор.
    f.prototype = p;                  // Записать в его свойство prototype
                                      // ссылку на объект p.
    return new f();                   // Использовать f() для создания
                                      // "наследника" объекта p.
}
```

---

[1] Дуглас Крокфорд (Douglas Crockford) считается первым, кто реализовал функцию, создающую объекты таким способом. См. *http://javascript.crockford.com/prototypal.html*.

Реализация функции `inherit()` приобретет больше смысла, как только мы познакомимся с конструкторами в главе 9. А пока просто считайте, что она возвращает новый объект, наследующий свойства объекта в аргументе. Обратите внимание, что функция `inherit()` не является полноценной заменой для `Object.create()`: она не позволяет создавать объекты без прототипа и не принимает второй необязательный аргумент, как `Object.create()`. Тем не менее мы будем использовать функцию `inherit()` во многих примерах в этой главе и в главе 9.

Функцию `inherit()` можно использовать, чтобы защитить объекты от непреднамеренного (не с целью нанести вред) изменения их библиотечными функциями, неподконтрольными вам. Вместо того чтобы передавать функции сам объект непосредственно, можно передать его наследника. Если функция прочитает свойства наследника, она получит унаследованные значения. Однако если она изменяет значения свойств, эти изменения коснутся только наследника и никак не отразятся на оригинальном объекте:

```
var o = { x: "не изменяйте это значение" };
library_function(inherit(o));      // Защита объекта o от непреднамеренного изменения
```

Чтобы понять принцип действия этого приема, необходимо знать, как производится чтение и запись значений свойств объектов в языке JavaScript. Об этом рассказывается в следующем разделе.

6.2. Получение и изменение свойств

Получить значение свойства можно с помощью операторов точки (`.`) и квадратных скобок (`[]`), описанных в разделе 4.4. Слева от оператора должно находиться выражение, возвращающее объект. При использовании оператора точки справа должен находиться простой идентификатор, соответствующий имени свойства. При использовании квадратных скобок в квадратных скобках должно указываться выражение, возвращающее строку, содержащую имя требуемого свойства:

```
var author = book.author;          // Получить свойство "author" объекта book.
var name = author.surname          // Получить свойство "surname" объекта author.
var title = book["main title"]     // Получить свойство "main title" объекта book.
```

Чтобы создать новое свойство или изменить значение существующего свойства, также используются операторы точки и квадратные скобки, как в операциях чтения значений свойств, но само выражение помещается уже слева от оператора присваивания:

```
book.edition = 6;                  // Создать свойство "edition" объекта book.
book["main title"] = "ECMAScript"; // Изменить значение свойства "main title".
```

В ECMAScript 3 идентификатор, следующий за точкой, не может быть зарезервированным словом: нельзя записать обращение к свойству `o.for` или `o.class`, потому что `for` является ключевым словом, а `class` – словом, зарезервированным для использования в будущем. Если объект имеет свойства, имена которых совпадают с зарезервированными словами, для доступа к ним необходимо использовать форму записи с квадратными скобками: `o["for"]` и `o["class"]`. Стандарт ECMAScript 5 ослабляет это требование (как это уже сделано в некоторых реализациях ECMAScript 3) и допускает возможность использования зарезервированных слов после оператора точки.

Выше уже говорилось, что при использовании формы записи с квадратными скобками выражение в скобках должно возвращать строку. Если быть более точными, это выражение должно возвращать строку или значение, которое может быть преобразовано в строку. В главе 7, например, мы увидим распространенный прием использования чисел в квадратных скобках.

6.2.1. Объекты как ассоциативные массивы

Как отмечалось выше, следующие два выражения возвращают одно и то же значение:

```
object.property
object["property"]
```

Первая форма записи, с использованием точки и идентификатора, напоминает синтаксис доступа к статическому полю структуры или объекта в языке C или Java. Вторая форма записи, с использованием квадратных скобок и строки, выглядит как обращение к элементу массива, но массива, который индексируется строками, а не числами. Такого рода массивы называются ассоциативными массивами (а также хешами и словарями). Объекты в языке JavaScript являются ассоциативными массивами, и в этом разделе объясняется, почему это так важно.

В C, C++, Java и других языках программирования со строгим контролем типов объект может иметь только фиксированное число свойств, а имена этих свойств должны определяться заранее. Поскольку JavaScript относится к языкам программирования со слабым контролем типов, данное правило в нем не действует: программы могут создавать любое количество свойств в любых объектах. Однако при использовании для обращения к свойству оператора точка (.) имя свойства определяется идентификатором. Идентификаторы должны вводиться в тексте программы буквально – это не тип данных, поэтому в программе невозможно реализовать вычисление идентификаторов.

Напротив, когда для обращения к свойствам объекта используется форма записи с квадратными скобками ([]), имя свойства определяется строкой. Строки в языке JavaScript являются типом данных, поэтому они могут создаваться и изменяться в ходе выполнения программы. Благодаря этому, например, в языке JavaScript имеется возможность писать такой программный код:

```
var addr = "";
for(i = 0; i < 4; i++)
    addr += customer["address" + i] + '\n';
```

Этот фрагмент читает и объединяет в одну строку значения свойств `address0`, `address1`, `address2` и `address3` объекта `customer`.

Этот короткий пример демонстрирует гибкость использования формы записи с квадратными скобками и строковыми выражениями для доступа к свойствам объекта. Пример выше можно переписать с использованием оператора точки, но иногда встречаются случаи, когда доступ к свойствам можно организовать только с помощью формы записи с квадратными скобками. Представим, например, что необходимо написать программу, использующую сетевые ресурсы для вычисления текущего значения инвестиций пользователя в акции. Программа должна позволять пользователю вводить имя каждой компании, акциями которой он владеет, а также количество акций каждой компании. Для хранения этих данных можно было бы создать объект с именем portfolio. Объект имеет по одному свойству для каждой компании. Имя свойства одновременно является названием компании, а значение свойства определяет количество акций этой компании. То есть если, к примеру, пользователь владеет 50 акциями компании IBM, свойство portfolio.ibm будет иметь значение 50.

Следующая функция, добавляющая информацию об очередном пакете акций, могла бы быть частью такой программы:

```
function addstock(portfolio, stockname, shares) {
    portfolio[stockname] = shares;
}
```

Поскольку пользователь вводит имена компаний во время выполнения, нет никакого способа заранее определить эти имена. А так как на момент создания программы имена свойств нам неизвестны, мы не можем использовать оператор точки (.) для доступа к свойствам объекта portfolio. Однако мы можем задействовать оператор [], потому что для обращения к свойствам он позволяет использовать строковые значения (которые являются динамическими и могут изменяться во время выполнения) вместо идентификаторов (которые являются статическими и должны жестко определяться в тексте программы).

В главе 5 был представлен цикл for/in (и еще раз мы встретимся с ним чуть ниже, в разделе 6.5). Мощь этой инструкции языка JavaScript становится особенно очевидной, когда она применяется для работы с ассоциативными массивами. Ниже показано, как можно использовать ее для вычисления суммарного объема инвестиций в portfolio:

```
function getvalue(portfolio) {
    var total = 0.0;
    for(stock in portfolio) {        // Для каждой компании в portfolio:
        var shares = portfolio[stock]; // получить количество акций
        var price = getquote(stock);  // отыскать стоимость одной акции
        total += shares * price;      // прибавить к суммарному значению
    }
    return total;                     // Вернуть сумму.
}
```

6.2.2. Наследование

Объекты в языке JavaScript обладают множеством «собственных свойств» и могут также наследовать множество свойств от объекта-прототипа. Чтобы разобраться в этом, необходимо внимательно изучить механизм доступа к свойствам. В примерах этого раздела для создания объектов с определенными прототипами используется функция inherit() из примера 6.1.

Предположим, что программа обращается к свойству x объекта o. Если объект o не имеет собственного свойства с таким именем, выполняется попытка отыскать свойство x в прототипе объекта o. Если объект-прототип не имеет собственного свойства с этим именем, но имеет свой прототип, выполняется попытка отыскать свойство в прототипе прототипа. Так продолжается до тех пор, пока не будет найдено свойство x или пока не будет достигнут объект, не имеющий прототипа. Как видите, атрибут prototype объекта создает цепочку, или связанный список объектов, от которых наследуются свойства.

```
var o = {}              // o наследует методы объекта Object.prototype
o.x = 1;                // и обладает собственным свойством x.
var p = inherit(o);     // p наследует свойства объектов o и Object.prototype
p.y = 2;                // и обладает собственным свойством y.
var q = inherit(p);     // q наследует свойства объектов p, o и Object.prototype
q.z = 3;                // и обладает собственным свойством z.
var s = q.toString();   // toString наследуется от Object.prototype
q.x + q.y               // => 3: x и y наследуются от o и p
```

Теперь предположим, что программа присваивает некоторое значение свойству x объекта o. Если объект o уже имеет собственное свойство (не унаследованное) с именем x, то операция присваивания просто изменит значение существующего свойства. В противном случае в объекте o будет создано новое свойство с именем x. Если прежде объект o наследовал свойство x, унаследованное свойство теперь окажется скрыто вновь созданным собственным свойством с тем же именем.

Операция присваивания значения свойству проверит наличие этого свойства в цепочке прототипов, чтобы убедиться в допустимости присваивания. Например, если объект o наследует свойство x, доступное только для чтения, то присваивание выполняться не будет. (Подробнее о том, когда свойство может устанавливаться, рассказывается в разделе 6.2.3.) Однако если присваивание допустимо, всегда создается или изменяется свойство в оригинальном объекте и никогда в цепочке прототипов. Тот факт, что механизм наследования действует при чтении свойств, но не действует при записи новых значений, является ключевой особенностью языка JavaScript, потому что она позволяет выборочно переопределять унаследованные свойства:

```
var unitcircle = { r:1 };       // Объект, от которого наследуется свойство
var c = inherit(unitcircle);    // c наследует свойство r
c.x = 1; c.y = 1;               // c определяет два собственных свойства
c.r = 2;                        // c переопределяет унаследованное свойство
unitcircle.r;                   // => 1: объект-прототип не изменился
```

Существует одно исключение из этого правила, когда операция присваивания значения свойству терпит неудачу или приводит к созданию/изменению свойства оригинального объекта. Если объект o наследует свойство x и доступ к этому свойству осуществляется посредством методов доступа (раздел 6.6), то вместо создания нового свойства x в объекте o производится вызов метода записи нового значения. Однако обратите внимание, что метод записи вызывается относительно объекта o, а не относительно прототипа, в котором определено это свойство, поэтому, если метод записи определяет какие-либо свойства, они будут созданы в объекте o, а цепочка прототипов опять останется неизменной.

6.2.3. Ошибки доступа к свойствам

Выражения обращения к свойствам не всегда возвращают или изменяют значение свойства. В этом разделе описываются ситуации, когда операции чтения или записи свойства терпят неудачу.

Попытка обращения к несуществующему свойству не считается ошибкой. Если свойство x не будет найдено среди собственных или унаследованных свойств объекта o, выражение обращения к свойству o.x вернет значение undefined. Напомню, что наш объект book имеет свойство с именем «sub-title», но не имеет свойства «subtitle»:

```
book.subtitle; // => undefined: свойство отсутствует
```

Однако попытка обратиться к свойству несуществующего объекта считается ошибкой. Значения null и undefined не имеют свойств, и попытки обратиться к свойствам этих значений считаются ошибкой. Продолжим пример, приведенный выше:

```
// Возбудит исключение TypeError. Значение undefined не имеет свойства length
var len = book.subtitle.length;
```

Если нет уверенности, что book и book.subtitle являются объектами (или ведут себя подобно объектам), нельзя использовать выражение book.subtitle.length, так как оно может возбудить исключение. Ниже демонстрируются два способа защиты против исключений подобного рода:

```
// Более наглядный и прямолинейный способ
var len = undefined;
if (book) {
    if (book.subtitle) len = book.subtitle.length;
}

// Более краткая и характерная для JavaScript альтернатива получения длины
// значения свойства subtitle
var len = book && book.subtitle && book.subtitle.length;
```

Чтобы понять, почему второе выражение позволяет предотвратить появление исключений TypeError, можете вернуться к описанию короткой схемы вычислений, используемой оператором &&, в разделе 4.10.1. Разумеется, попытка установить значение свойства для значения null или undefined также вызывает исключение TypeError.

Попытки установить значение свойства для других значений не всегда оканчиваются успехом: некоторые свойства доступны только для чтения и не позволяют изменять их значения. Кроме того, некоторые объекты не позволяют добавлять в них новые свойства. Однако самое интересное, что подобные неудачи, как правило, не приводят к возбуждению исключения:

```
// Свойства prototype встроенных конструкторов доступны только для чтения.
Object.prototype = 0; // Присваивание не возбудит исключения;
                      // значение Object.prototype не изменится
```

Этот исторически сложившийся недостаток JavaScript исправлен в строгом режиме, определяемом стандартом ECMAScript 5. Все неудачные попытки изменить значение свойства в строгом режиме приводят к исключению TypeError.

Правила, позволяющие определить, когда попытка выполнить операцию присваивания завершится успехом, а когда неудачей, просты и понятны, но их сложно выразить в достаточно краткой форме. Попытка присвоить значение свойству p объекта o потерпит неудачу в следующих случаях:

- Объект o имеет собственное свойство p, доступное только для чтения: нельзя изменить значение свойства, доступного только для чтения. (Обратите, однако, внимание на метод defineProperty(), который представляет собой исключение, позволяющее изменять значения настраиваемых свойств, доступных только для чтения.)

- **Объект `o` имеет унаследованное свойство `p`, доступное только для чтения: унаследованные свойства, доступные только для чтения, невозможно переопределить собственными свойствами с теми же именами.**
- **Объект `o` не имеет собственного свойства `p`; объект `o` не наследует свойство `p` с методами доступа и атрибут `extensible` (раздел 6.8.3) объекта `o` имеет значение `false`. Если свойство `p` отсутствует в объекте `o` и для него не определен метод записи, то операция присваивания попытается добавить свойство `p` в объект `o`. Но поскольку объект `o` не допускает возможность расширения, то попытка добавить в него новое свойство потерпит неудачу.**

6.3. Удаление свойств

Оператор `delete` (раздел 4.13.3) удаляет свойство из объекта. Его единственный операнд должен быть выражением обращения к свойству. Может показаться удивительным, но оператор `delete` не оказывает влияния на значение свойства – он оперирует самим свойством:

```
delete book.author;       // Теперь объект book не имеет свойства author.
delete book["main title"]; // Теперь он не имеет свойства "main title".
```

Оператор `delete` удаляет только собственные свойства и не удаляет унаследованные. (Чтобы удалить унаследованное свойство, необходимо удалять его в объекте-прототипе, в котором оно определено. Такая операция затронет все объекты, наследующие этот прототип.)

Выражение `delete` возвращает значение `true` в случае успешного удаления свойства или когда операция удаления не привела к изменению объекта (например, при попытке удалить несуществующее свойство). Выражение `delete` также возвращает `true`, когда этому оператору передается выражение, не являющееся выражением обращения к свойству:

```
o = {x:1};          // o имеет собственное свойство x и наследует toString
delete o.x;         // Удалит x и вернет true
delete o.x;         // Ничего не сделает (x не существует) и вернет true
delete o.toString;  // Ничего не сделает (toString не собственное свойство) и вернет true
delete 1;           // Бессмысленно, но вернет true
```

Оператор `delete` не удаляет ненастраиваемые свойства, атрибут `configurable` которых имеет значение `false`. (Однако он может удалять настраиваемые свойства нерасширяемых объектов.) Ненастраиваемыми являются свойства встроенных объектов, а также свойства глобального объекта, созданные с помощью инструкций объявления переменных и функций. Попытка удалить ненастраиваемое свойство в строгом режиме вызывает исключение `TypeError`. В нестрогом режиме (и в реализациях ECMAScript 3) в таких случаях оператор `delete` просто возвращает `false`:

```
delete Object.prototype; // Удаление невозможно - ненастраиваемое свойство
var x = 1;               // Объявление глобальной переменной
delete this.x;           // Это свойство нельзя удалить
function f() {}          // Объявление глобальной функции
delete this.f;           // Это свойство также нельзя удалить
```

При удалении настраиваемых свойств глобального объекта в нестрогом режиме допускается опускать ссылку на глобальный объект и передавать оператору `delete` только имя свойства:

```
this.x = 1;   // Создать настраиваемое глобальное свойство (без var)
delete x;     // И удалить его
```

Однако в строгом режиме оператор `delete` возбуждает исключение `SyntaxError`, если его операндом является неквалифицированный идентификатор, такой как `x`, поэтому необходимо указывать явное выражение обращения к свойству:

```
delete x;     // В строгом режиме возбудит исключение SyntaxError
delete this.x; // Такой способ работает
```

6.4. Проверка существования свойств

Объекты в языке JavaScript можно рассматривать как множества свойств, и нередко бывает полезно иметь возможность проверить принадлежность к множеству – проверить наличие в объекте свойства с данным именем. Выполнить такую проверку можно с помощью оператора `in`, с помощью методов `hasOwnProperty()` и `propertyIsEnumerable()` или просто обратившись к свойству.

Оператор `in` требует, чтобы в левом операнде ему было передано имя свойства (в виде строки) и объект в правом операнде. Он возвращает `true`, если объект имеет собственное или унаследованное свойство с этим именем:

```
var o = { x: 1 }
"x" in o;        // true: o имеет собственное свойство "x"
"y" in o;        // false: o не имеет свойства "y"
"toString" in o; // true: o наследует свойство toString
```

Метод `hasOwnProperty()` объекта проверяет, имеет ли объект собственное свойство с указанным именем. Для наследуемых свойств он возвращает `false`:

```
var o = { x: 1 }
o.hasOwnProperty("x");        // true: o имеет собственное свойство x
o.hasOwnProperty("y");        // false: не имеет свойства y
o.hasOwnProperty("toString"); // false: toString - наследуемое свойство
```

Метод `propertyIsEnumerable()` накладывает дополнительные ограничения по сравнению с `hasOwnProperty()`. Он возвращает `true`, только если указанное свойство является собственным свойством, атрибут `enumerable` которого имеет значение `true`. Свойства встроенных объектов не являются перечислимыми. Свойства, созданные обычной программой на языке JavaScript, являются перечислимыми, если не был использован один из методов ECMAScript 5, представленных ниже, которые делают свойства неперечислимыми.

```
var o = inherit({ y: 2 });
o.x = 1;
o.propertyIsEnumerable("x"); // true: o имеет собств. перечислимое свойство x
o.propertyIsEnumerable("y"); // false: y - унаследованное свойство, не собств.
Object.prototype.propertyIsEnumerable("toString"); // false: неперечислимое
```

Часто вместо оператора `in` достаточно использовать простое выражение обращения к свойству и использовать оператор `!==` для проверки на неравенство значению `undefined`:

```
var o = { x: 1 }
o.x !== undefined;        // true: o имеет свойство x
o.y !== undefined;        // false: o не имеет свойства y
o.toString !== undefined; // true: o наследует свойство toString
```

Однако оператор `in` отличает ситуации, которые неотличимы при использовании представленного выше приема на основе обращения к свойству. Оператор `in` отличает отсутствие свойства от свойства, имеющего значение `undefined`. Взгляните на следующий пример:

```
var o = { x: undefined }  // Свойству явно присвоено значение undefined
o.x !== undefined         // false: свойство имеется, но со значением undefined
o.y !== undefined         // false: свойство не существует
"x" in o                  // true: свойство существует
"y" in o                  // false: свойство не существует
delete o.x;               // Удалить свойство x
"x" in o                  // false: оно больше не существует
```

Обратите внимание, что в примере выше использован оператор `!==`, а не `!=`. Операторы `!==` и `===` отличают значения `undefined` и `null`, хотя иногда в этом нет необходимости:

```
// Если o имеет свойство x, значение которого отлично от null и undefined,
// то удвоить это значение.
if (o.x != null) o.x *= 2;

// Если o имеет свойство x, значение которого не может быть преобразовано в false,
// то удвоить это значение. Если x имеет значение undefined, null, false, "", 0 или NaN,
// оставить его в исходном состоянии.
if (o.x) o.x *= 2;
```

6.5. Перечисление свойств

Вместо проверки наличия отдельных свойств иногда бывает необходимо обойти все имеющиеся свойства или получить список всех свойств объекта. Обычно для этого используется цикл `for/in`, однако стандарт ECMAScript 5 предоставляет две удобные альтернативы.

Инструкция цикла `for/in` рассматривалась в разделе 5.5.4. Она выполняет тело цикла для каждого перечислимого свойства (собственного или унаследованного) указанного объекта, присваивая имя свойства переменной цикла. Встроенные методы, наследуемые объектами, являются неперечислимыми, а свойства, добавляемые в объекты вашей программой, являются перечислимыми (если только не использовались функции, описываемые ниже, позволяющие сделать свойства неперечислимыми). Например:

```
var o = {x:1, y:2, z:3};          // Три собственных перечислимых свойства
o.propertyIsEnumerable("toString") // => false: неперечислимое
for(p in o)                       // Цикл по свойствам
    console.log(p);               // Выведет x, y и z, но не toString
```

Некоторые библиотеки добавляют новые методы (или другие свойства) в объект `Object.prototype`, чтобы они могли быть унаследованы и быть доступны всем объектам. Однако до появления стандарта ECMAScript 5 отсутствовала возможность сделать эти дополнительные методы неперечислимыми, поэтому они оказывались доступными для перечисления в циклах `for/in`. Чтобы решить эту проблему, может потребоваться фильтровать свойства, возвращаемые циклом `for/in`. Ниже приводятся два примера реализации такой фильтрации:

```
for(p in o) {
    if (!o.hasOwnProperty(p)) continue;       // Пропустить унаследованные свойства
}

for(p in o) {
    if (typeof o[p] === "function") continue; // Пропустить методы
}
```

В примере 6.2 определяются вспомогательные функции, использующие цикл `for/in` для управления свойствами объектов. Функция `extend()`, в частности, часто используется в библиотеках JavaScript.[1]

Пример 6.2. Вспомогательные функции, используемые для перечисления свойств объектов

```
/*
 * Копирует перечислимые свойства из объекта p в объект o и возвращает o.
 * Если o и p имеют свойства с одинаковыми именами, значение свойства
 * в объекте o затирается значением свойства из объекта p.
 * Эта функция не учитывает наличие методов доступа и не копирует атрибуты.
 */
function extend(o, p) {
    for(prop in p) {            // Для всех свойств в p.
        o[prop] = p[prop];      // Добавить свойство в o.
    }
    return o;
}

/*
 * Копирует перечислимые свойства из объекта p в объект o и возвращает o.
 * Если o и p имеют свойства с одинаковыми именами, значение свойства
 * в объекте o остается неизменным.
 * Эта функция не учитывает наличие методов доступа и не копирует атрибуты.
 */
function merge(o, p) {
    for(prop in p) {                            // Для всех свойств в p.
        if (o.hasOwnProperty[prop]) continue;   // Кроме имеющихся в o.
        o[prop] = p[prop];                      // Добавить свойство в o.
    }
    return o;
}

/*
 * Удаляет из объекта o свойства, отсутствующие в объекте p.
 * Возвращает o.
 */
```

---

[1] Функция `extend()`, представленная здесь, реализована правильно, но она не компенсирует хорошо известную проблему в Internet Explorer. Более надежная версия функции `extend()` будет представлена в примере 8.3.

```
function restrict(o, p) {
    for(prop in o) {                      // Для всех свойств в o
        if (!(prop in p)) delete o[prop]; // Удалить, если отсутствует в p
    }
    return o;
}

/*
 * Удаляет из объекта o свойства, присутствующие в объекте p. Возвращает o.
 */
function subtract(o, p) {
    for(prop in p) {           // Для всех свойств в p
        delete o[prop];        // Удалить из o (удаление несуществующих
                               // свойств можно выполнять без опаски)
    }
    return o;
}
/*
 * Возвращает новый объект, содержащий свойства, присутствующие хотя бы в одном
 * из объектов, o или p. Если оба объекта, o и p, имеют свойства с одним
 * и тем же именем, используется значение свойства из объекта p.
 */
function union(o,p) { return extend(extend({},o), p); }

/*
 * Возвращает новый объект, содержащий свойства, присутствующие сразу в обоих
 * объектах, o или p. Результат чем-то напоминает пересечение o и p,
 * но значения свойств объекта p отбрасываются
 */
function intersection(o,p) { return restrict(extend({}, o), p); }

/*
 * Возвращает массив имен собственных перечислимых свойств объекта o.
 */
function keys(o) {
    if (typeof o !== "object") throw TypeError(); // Арг. должен быть объектом
    var result = [];                    // Возвращаемый массив
    for(var prop in o) {                // Для всех перечислимых свойств
        if (o.hasOwnProperty(prop))     // Если это собственное свойство,
            result.push(prop);          // добавить его в массив array.
    }
    return result;                      // Вернуть массив.
}
```

В дополнение к циклу `for/in` стандарт ECMAScript 5 определяет две функции, перечисляющие имена свойств. Первая из них, `Object.keys()`, возвращает массив имен собственных перечислимых свойств объекта. Она действует аналогично функции `keys()` из примера 6.2.

Вторая функция ECMAScript 5, выполняющая перечисление свойств, – `Object.getOwnPropertyNames()`. Она действует подобно функции `Object.keys()`, но возвращает имена всех собственных свойств указанного объекта, а не только перечислимые. В реализациях ECMAScript 3 отсутствует возможность реализовать подобные функции, потому что ECMAScript 3 не предусматривает возможность получения неперечислимых свойств объекта.

6.6. Методы чтения и записи свойств

Выше уже говорилось, что свойство объекта имеет имя, значение и набор атрибутов. В ECMAScript 5[1] значение может замещаться одним или двумя методами, известными как методы *чтения* (getter) и *записи* (setter). Свойства, для которых определяются методы чтения и записи, иногда называют ***свойствами с методами доступа***, чтобы отличать их от ***свойств с данными***, представляющих простое значение.

Когда программа пытается получить значение свойства с методами доступа, интерпретатор вызывает метод чтения (без аргументов). Возвращаемое этим методом значение становится значением выражения обращения к свойству. Когда программа пытается записать значение в свойство, интерпретатор вызывает метод записи, передавая ему значение, находящее справа от оператора присваивания. Этот метод отвечает за «установку» значения свойства. Значение, возвращаемое методом записи, игнорируется.

В отличие от свойств с данными, свойства с методами доступа не имеют атрибута `writable`. Если свойство имеет оба метода, чтения и записи, оно доступно для чтения/записи. Если свойство имеет только метод чтения, оно доступно только для чтения. А если свойство имеет только метод записи, оно доступно только для записи (такое невозможно для свойств с данными) и попытки прочитать значение такого свойства всегда будут возвращать `undefined`.

Самый простой способ определить свойство с методами доступа заключается в использовании расширенного синтаксиса определения литералов объектов:

```
var o = {
    // Обычное свойство с данными
    data_prop: value,

    // Свойство с методами доступа определяется как пара функций
    get accessor_prop() { /* тело функции */ },
    set accessor_prop(value) { /* тело функции */ }
};
```

Свойства с методами доступа определяются как одна или две функции, имена которых совпадают с именем свойства и с заменой ключевого слова `function` на `get` и/или `set`. Обратите внимание, что не требуется использовать двоеточие для отделения имени свойства от функции, управляющей доступом к свойству, но попрежнему необходимо использовать запятую после тела функции, чтобы отделить метод от других методов или свойств с данными. Для примера рассмотрим следующий объект, представляющий Декартовы координаты точки на плоскости. Для представления координат X и Y в нем имеются обычные свойства с данными, а также свойства с методами доступа, позволяющие получить эквивалентные полярные координаты точки:

```
var p = {
    // x и y - обычные свойства с данными, доступные для чтения/записи.
    x: 1.0,
```

---

[1] И в последних версиях реализации стандарта ECMAScript 3 в основных броузерах, кроме IE.

```
        y: 1.0,

        // r - доступное для чтения/записи свойство с двумя методами доступа.
        // Не забывайте добавлять запятые после методов доступа.
        get r() { return Math.sqrt(this.x*this.x + this.y*this.y); },
        set r(newvalue) {
            var oldvalue = Math.sqrt(this.x*this.x + this.y*this.y);
            var ratio = newvalue/oldvalue;
            this.x *= ratio;
            this.y *= ratio;
        },

        // theta - доступное только для чтения свойство с единственным методом чтения.
        get theta() { return Math.atan2(this.y, this.x); }
    };
```

Обратите внимание на использование ключевого слова `this` в методах чтения и записи выше. Интерпретатор будет вызывать эти функции, как методы объекта, в котором они определены, т. е. в теле функции `this` будет ссылаться на объект точки. Благодаря этому метод чтения свойства `r` может ссылаться на свойства `x` и `y`, как `this.x` и `this.y`. Подробнее о методах и ключевом слове `this` рассказывается в разделе 8.2.2.

Свойства с методами доступа наследуются так же, как обычные свойства с данными, поэтому объект `p`, определенный выше, можно использовать как прототип для других объектов точек. В новых объектах можно определять собственные свойства `x` и `y`, и они будут наследовать свойства `r` и `theta`:

```
var q = inherit(p);      // Создать новый объект, наследующий методы доступа
q.x = 1; q.y = 1;        // Создать собственные свойства с данными в объекте q
console.log(q.r);        // И использовать унаследованные свойства
console.log(q.theta);    // с методами доступа
```

Фрагмент выше использует свойства с методами доступа для определения API, обеспечивающего представление единого набора данных в двух системах координат (Декартовой и полярной). Еще одной причиной использования свойств с методами доступа может быть необходимость проверки значения перед записью и возврат разных значений при каждом чтении свойства:

```
// Этот объект генерирует последовательность увеличивающихся чисел
var serialnum = {
    // Это свойство с данными хранит следующее число в последовательности.
    // Знак $ в имени свойства говорит о том, что оно является частным.
    $n: 0,

    // Возвращает текущее значение и увеличивает его
    get next() { return this.$n++; },

    // Устанавливает новое значение n, но только если оно больше текущего
    set next(n) {
        if (n >= this.$n) this.$n = n;
        else throw "число может быть только увеличено ";
    }
};
```

Наконец, ниже приводится еще один пример использования метода чтения для реализации свойства с «таинственным» поведением.

```
// Этот объект имеет свойства с методами доступа, при обращении к которым возвращаются
// случайные числа. Например, каждый раз при вычислении выражения "random.octet"
// будет возвращаться случайное число в диапазоне от 0 до 255.
var random = {
    get octet() { return Math.floor(Math.random()*256); },
    get uint16() { return Math.floor(Math.random()*65536); },
    get int16() { return Math.floor(Math.random()*65536)-32768; }
};
```

В этом разделе было показано, как определять свойства с методами доступа при создании нового объекта с помощью литерала. В следующем разделе будет показано, как добавлять свойства с методами доступа в существующие объекты.

6.7. Атрибуты свойств

Помимо имени и значения свойства обладают атрибутами, определяющими их доступность для записи, перечисления и настройки. В ECMAScript 3 не предусматривается возможность изменения атрибутов: все свойства, создаваемые программами, выполняющимися под управлением реализации ECMAScript 3, доступны для записи, перечисления и настройки, и нет никакой возможности изменить эти атрибуты. Данный раздел описывает прикладной интерфейс (API), определяемый стандартом ECMAScript 5 для получения и изменения атрибутов свойств. Данный API имеет особое значение для разработчиков библиотек, потому что он позволяет:

- добавлять методы в объекты-прототипы и делать их неперечислимыми, подобно встроенным методам;
- «ограничивать» возможности объектов за счет определения свойств, которые не могут изменяться или удаляться.

Для целей данного раздела мы будем рассматривать методы чтения и записи свойств с методами как атрибуты свойств. Следуя этой логике, можно даже сказать, что значение свойства с данными также является атрибутом. Таким образом, свойства имеют имя и четыре атрибута. Четырьмя атрибутами свойств с данными являются: *значение* (`value`), *признак доступности для записи* (`writable`), *признак доступности для перечисления* (`enumerable`) и *признак доступности для настройки* (`configurable`). В свойствах с методами доступа отсутствуют атрибуты `value` и `writable`: их доступность для записи определяется наличием или отсутствием метода записи. Поэтому четырьмя атрибутами свойств с методами доступа являются: *метод чтения* (`get`), *метод записи* (`set`), *признак доступности для перечисления* (`enumerable`) и *признак доступности для настройки* (`configurable`).

Методы получения и записи значений атрибутов свойств, предусмотренные стандартом ECMAScript 5, используют объект, называемый *дескриптором свойства* (property descriptor), представляющий множество из четырех атрибутов. Объект дескриптора свойства обладает свойствами, имена которых совпадают с именами атрибутов свойства, которое он описывает. То есть объекты-дескрипторы свойств с данными имеют свойства с именами `value`, `writable`, `enumerable` и `configurable`. А дескрипторы свойств с методами доступа вместо свойств `value` и `writable` имеют свойства `get` и `set`. Свойства `writable`, `enumerable` и `configurable` являются логическими значениями, а свойства `get` и `set` – функциями.

Получить дескриптор свойства требуемого объекта можно вызовом `Object.getOwnPropertyDescriptor()`:

```
// Вернет {value: 1, writable:true, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor({x:1}, "x");

// Теперь получим свойство octet объекта random, объявленного выше.
// Вернет { get: /*func*/, set:undefined, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor(random, "octet");

// Вернет undefined для унаследованных и несуществующих свойств.
Object.getOwnPropertyDescriptor({}, "x");        // undefined, нет такого свойства
Object.getOwnPropertyDescriptor({}, "toString"); // undefined, унаследованное
```

Как можно заключить из названия метода, `Object.getOwnPropertyDescriptor()` работает только с собственными свойствами. Чтобы получить атрибуты унаследованного свойства, необходимо явно выполнить обход цепочки прототипов (смотрите описание `Object.getPrototypeOf()` в разделе 6.8.1).

Чтобы изменить значение атрибута свойства или создать новое свойство с заданными значениями атрибутов, следует вызвать метод `Object.defineProperty()`, передав ему объект, в котором требуется выполнить изменения, имя создаваемого или изменяемого свойства и объект дескриптора свойства:

```
var o = {};            // Создать пустой объект без свойств
// Создать неперечислимое простое свойство x со значением 1.
Object.defineProperty(o, "x", { value : 1,
                                writable: true,
                                enumerable: false,
                                configurable: true});

// Убедиться, что свойство создано и является неперечислимым
o.x;                   // => 1
Object.keys(o)         // => []

// Теперь сделать свойство x доступным только для чтения
Object.defineProperty(o, "x", { writable: false });

// Попытаться изменить значение свойства
o.x = 2;               // Неудача, в строгом режиме возбудит TypeError
o.x                    // => 1

// Свойство все еще доступно для настройки, его значение можно изменить так:
Object.defineProperty(o, "x", { value: 2 });
o.x                    // => 2

// Теперь превратить простое свойство в свойство с методами доступа
Object.defineProperty(o, "x", { get: function() { return 0; } });
o.x                    // => 0
```

Дескриптор свойства, передаваемый методу `Object.defineProperty()`, необязательно должен иметь все четыре атрибута. При создании нового свойства отсутствующие атрибуты получат значение `false` или `undefined`. При изменении существующего свойства для отсутствующих атрибутов будут сохранены текущие значения. Обратите внимание, что этот метод изменяет существующее собственное свойство или создает новое собственное свойство – он не изменяет унаследованные свойства.

Если возникнет необходимость создать или изменить сразу несколько свойств, можно воспользоваться методом `Object.define Properties()`. Первым аргументом ему передается объект, который требуется изменить. Вторым аргументом – объект, отображающий имена создаваемых или модифицируемых свойств в дескрипторы этих свойств. Например:

```
var p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable:true, configurable:true },
    y: { value: 1, writable: true, enumerable:true, configurable:true },
    r: {
        get: function() { return Math.sqrt(this.x*this.x + this.y*this.y) },
        enumerable:true,
        configurable:true
    }
});
```

В этом примере все начинается с пустого объекта, в который затем добавляются два свойства с данными и одно свойство с методами доступа, доступное только для чтения. Он опирается на тот факт, что `Object.defineProperties()` возвращает модифицированный объект (подобно методу `Object.defineProperty()`).

С методом `Object.create()`, определяемым стандартом ECMAScript 5, мы познакомились в разделе 6.1, где узнали, что первым аргументом этому методу передается объект, который будет служить прототипом для вновь созданного объекта. Этот метод также принимает второй необязательный аргумент, такой же, как и второй аргумент метода `Object.defineProperties()`. Если методу `Object.create()` передать множество дескрипторов свойств, они будут использованы для создания свойств нового объекта.

Методы `Object.defineProperty()` и `Object.defineProperties()` возбуждают исключение `TypeError`, когда создание или изменение свойств запрещено. Например, при попытке добавить новое свойство в нерасширяемый объект (раздел 6.8.3). Другие причины, по которым эти методы могут возбудить исключение `TypeError`, имеют непосредственное отношение к атрибутам. Атрибут `writable` контролирует попытки изменить атрибут `value`. А атрибут `configurable` контролирует попытки изменить другие атрибуты (а также определяет возможность удаления свойства). Однако все не так просто. Например, значение свойства, доступного только для чтения, можно изменить, если это свойство доступно для настройки. Кроме того, свойство, доступное только для чтения, можно сделать доступным для записи, даже если это свойство недоступно для настройки. Ниже приводится полный перечень правил. Вызовы `Object.defineProperty()` или `Object.defineProperties()`, нарушающие их, возбуждают исключение `TypeError`:

- Если объект нерасширяемый, можно изменить существующие собственные свойства этого объекта, но нельзя добавить в него новые свойства.
- Если свойство недоступно для настройки, нельзя изменить его атрибуты `configurable` и `enumerable`.
- Если свойство с методами доступа недоступно для настройки, нельзя изменить его методы чтения и записи и нельзя превратить его в простое свойство с данными.
- Если свойство с данными недоступно для настройки, нельзя превратить его в свойство с методами доступа.

- Если свойство с данными недоступно для настройки, нельзя изменить значение его атрибута `writable` с `false` на `true`, но его можно изменить с `true` на `false`.
- Если свойство с данными недоступно для настройки и для записи, нельзя изменить его значение. Однако изменить значение свойства, недоступного для записи можно, если оно доступно для настройки (потому что свойство можно сделать доступным для записи, изменить его значение и затем опять сделать свойство доступным только для чтения).

Пример 6.2 включает функцию `extend()`, которая копирует свойства из одного объекта в другой. Эта функция просто копирует имена и значения свойств и игнорирует их атрибуты. Кроме того, она не копирует методы чтения и записи из свойств с методами доступа, а просто преобразует их в свойства со статическими данными. В примере 6.3 показана новая версия `extend()`, которая копирует все атрибуты свойств с помощью `Object.getOwnPropertyDescriptor()` и `Object.defineProperty()`. Но на этот раз данная версия оформлена не как функция, а как новый метод объекта и добавляется в `Object.prototype` как свойство, недоступное для перечисления.

Пример 6.3. Копирование атрибутов свойств

```
/*
 * Добавляет неперечислимый метод extend() в Object.prototype.
 * Этот метод расширяет объекты возможностью копирования свойств из объекта,
 * переданного в аргументе. Этот метод копирует не только значение свойств,
 * но и все их атрибуты. Из объекта в аргументе копируются все собственные
 * свойства (даже недоступные для перечисления), за исключением одноименных
 * свойств, имеющихся в текущем объекте.
 */
Object.defineProperty(Object.prototype,
    "extend",                  // Определяется Object.prototype.extend
    {
        writable: true,
        enumerable: false,     // Сделать неперечислимым
        configurable: true,
        value: function(o) {   // Значением свойства является данная функция
            // Получить все собственные свойства, даже неперечислимые
            var names = Object.getOwnPropertyNames(o);
            // Обойти их в цикле
            for(var i = 0; i < names.length; i++) {
                // Пропустить свойства, уже имеющиеся в данном объекте
                if (names[i] in this) continue;
                // Получить дескриптор свойства из o
                var desc = Object.getOwnPropertyDescriptor(o,names[i]);
                // Создать с его помощью свойство в данном объекте
                Object.defineProperty(this, names[i], desc);
            }
        }
    });
```

6.7.1. Устаревшие приемы работы с методами чтения и записи

Синтаксис определения свойств с методами доступа в литералах объектов, описанный разделе 6.6, позволяет определять свойства с методами в новых объектах, но он не дает возможности получать методы чтения и записи и добавлять новые свойства с методами доступа к существующим объектам. В ECMAScript 5 для этих целей можно использовать `Object.getOwnPropertyDescriptor()` и `Object.defineProperty()`.

Большинство реализаций JavaScript (за исключением веб-броузера IE) поддерживали синтаксис `get` и `set` в литералах объектов еще до принятия стандарта ECMAScript 5. Эти реализации поддерживают нестандартный, устаревший API для получения и назначения методов чтения и записи. Этот API состоит из четырех методов, доступных во всех объектах. `__lookupGetter__()` и `__lookupSetter__()` возвращают методы чтения и записи для указанного свойства. А методы `__defineGetter__()` и `__defineSetter__()` позволяют определить метод чтения или записи: в первом аргументе они принимают имя свойства, а во втором – метод чтения или записи. Имена всех этих методов начинаются и оканчиваются двумя символами подчеркивания, чтобы показать, что они являются нестандартными методами. Эти нестандартные методы не описываются в справочном разделе.

6.8. Атрибуты объекта

Все объекты имеют атрибуты `prototype`, `class` и `extensible`. Все эти атрибуты описываются в подразделах ниже; в них также рассказывается, как получать и изменять значения атрибутов (если это возможно).

6.8.1. Атрибут prototype

Атрибут `prototype` объекта определяет объект, от которого наследуются свойства. (Дополнительные сведения о прототипах и наследовании прототипов приводятся в разделах 6.1.3 и 6.2.2.) Этот атрибут играет настолько важную роль, что обычно мы будем говорить о нем как о «прототипе объекта `o`», а не как об «атрибуте `prototype` объекта `o`». Кроме того, важно понимать, что когда в программном коде встречается ссылка `prototype`, она обозначает обычное свойство объекта, а не атрибут `prototype`.

Атрибут `prototype` устанавливается в момент создания объекта. В разделе 6.1.3 уже говорилось, что для объектов, созданных с помощью литералов, прототипом является `Object.prototype`. Прототипом объекта, созданного с помощью оператора `new`, является значение свойства `prototype` конструктора. А прототипом объекта, созданного с помощью `Object.create()`, становится первый аргумент этой функции (который может иметь значение `null`).

Стандартом ECMAScript 5 предусматривается возможность определить прототип любого объекта, если передать его методу `Object.getPrototypeOf()`. В ECMAScript 3 отсутствует эквивалентная функция, но зачастую определить прототип объекта `o` можно с помощью выражения `o.constructor.prototype`. Объекты, созданные с помощью оператора `new`, обычно наследуют свойство `constructor`, ссылающееся на функцию-конструктор, использованную для создания объекта. И как уже говорилось выше, функции-конструкторы имеют свойство `prototype`, которое определяет прототип объектов, созданных с помощью этого конструктора. Подробнее об этом рассказывается в разделе 9.2, где также объясняется, почему этот метод определения прототипа объекта не является достаточно надежным. Обратите внимание, что объекты, созданные с помощью литералов объектов или `Object.create()`, получают свойство `constructor`, ссылающееся на конструктор `Object()`. Таким образом, `constructor.prototype` ссылается на истинный прототип для литералов объектов, но обычно это не так для объектов, созданных вызовом `Object.create()`.

Чтобы определить, является ли один объект прототипом (или звеном в цепочке прототипов) другого объекта, следует использовать метод `isPrototypeOf()`. Чтобы узнать, является ли `p` прототипом `o`, нужно записать выражение `p.isPrototypeOf(o)`. Например:

```
var p = {x:1};                    // Определить объект-прототип.
var o = Object.create(p);         // Создать объект с этим прототипом.
p.isPrototypeOf(o)                // => true: o наследует p
Object.prototype.isPrototypeOf(p) // => true: p наследует Object.prototype
```

Обратите внимание, что `isPrototypeOf()` по своему действию напоминает оператор `instanceof` (раздел 4.9.4).

В реализации JavaScript компании Mozilla (первоначально созданной в Netscape) значение атрибута `prototype` доступно через специальное свойство `__proto__`, которое можно использовать напрямую для определения и установки прототипа любого объекта. Использование свойства `__proto__` ухудшает переносимость: оно отсутствует (и, вероятно, никогда не появится) в реализациях броузеров IE или Opera, хотя в настоящее время оно поддерживается броузерами Safari и Chrome. Версии Firefox, реализующие стандарт ECMAScript 5, все еще поддерживают свойство `__proto__`, но не позволяют изменять прототип нерасширяемых объектов.

6.8.2. Атрибут class

Атрибут `class` объекта – это строка, содержащая информацию о типе объекта. Ни в ECMAScript 3, ни в ECMAScript 5 не предусматривается возможность изменения этого атрибута и предоставляются лишь косвенные способы определения его значения. По умолчанию метод `toString()` (наследуемый от `Object.prototype`) возвращает строку вида:

```
[object class]
```

Поэтому, чтобы определить класс объекта, можно попробовать вызвать метод `toString()` этого объекта и извлечь из результата подстроку с восьмого по предпоследний символ. Вся хитрость состоит в том, что многие методы наследуют другие, более полезные реализации метода `toString()`, и чтобы вызвать нужную версию `toString()`, необходимо выполнить косвенный вызов с помощью метода `Function.call()` (раздел 8.7.3). В примере 6.4 определяется функция, возвращающая класс любого объекта, переданного ей.

Пример 6.4. Функция classof()

```
function classof(o) {
    if (o === null) return "Null";
    if (o === undefined) return "Undefined";
    return Object.prototype.toString.call(o).slice(8,-1);
}
```

Этой функции `classof()` можно передать любое значение, допустимое в языке JavaScript. Числа, строки и логические значения действуют подобно объектам, когда относительно них вызывается метод `toString()`, а значения `null` и `undefined` обрабатываются особо. (В ECMAScript 5 особая обработка не требуется.) Объекты, созданные с помощью встроенных конструкторов, таких как `Array` и `Date`, имеют атрибут `class`, значение которого совпадает с именами их конструкторов. Объекты среды выполнения обычно также получают осмысленное значение атрибута `class`, однако это зависит от реализации. Объекты, созданные с помощью литералов или вызовом `Object.create`, получают атрибут `class` со значением «Object». Если вы определите свой конструктор, все объекты, созданные с его помощью, получат атрибут `class` со значением «Object»: нет никакого способа установить иное значение в атрибуте `class` для собственных классов объектов:

```
classof(null)         // => "Null"
classof(1)            // => "Number"
classof("")           // => "String"
classof(false)        // => "Boolean"
classof({})           // => "Object"
classof([])           // => "Array"
classof(/./)          // => "Regexp"
classof(new Date())   // => "Date"
classof(window)       // => "Window" (объект клиентской среды выполнения)
function f() {};      // Определение собственного конструктора
classof(new f());     // => "Object"
```

6.8.3. Атрибут extensible

Атрибут `extensible` объекта определяет, допускается ли добавлять в объект новые свойства. В ECMAScript 3 все встроенные и определяемые пользователем объекты неявно допускали возможность расширения, а расширяемость объектов среды выполнения определялась каждой конкретной реализацией. В ECMAScript 5 все встроенные и определяемые пользователем объекты являются расширяемыми, если они не были преобразованы в нерасширяемые объекты, а расширяемость объектов среды выполнения по-прежнему определяется каждой конкретной реализацией.

Стандарт ECMAScript 5 определяет функции для получения и изменения признака расширяемости объекта. Чтобы определить, допускается ли расширять объект, его следует передать методу `Object.isExtensible()`. Чтобы сделать объект нерасширяемым, его нужно передать методу `Object.preventExtensions()`. Обратите внимание, что после того как объект будет сделан нерасширяемым, его нельзя снова сделать расширяемым. Отметьте также, что вызов `preventExtensions()` оказывает влияние только на расширяемость самого объекта. Если новые свойства добавить в прототип нерасширяемого объекта, нерасширяемый объект унаследует эти новые свойства.

Назначение атрибута `extensible` заключается в том, чтобы дать возможность «фиксировать» объекты в определенном состоянии, запретив внесение изменений. Атрибут объектов `extensible` часто используется совместно с атрибутами свойств `configurable` и `writable`, поэтому в ECMAScript 5 определяются функции, упрощающие одновременную установку этих атрибутов.

Метод `Object.seal()` действует подобно методу `Object.preventExtensions()`, но он не только делает объект нерасширяемым, но и делает все свойства этого объекта недоступными для настройки. То есть в объект нельзя будет добавить новые свойства, а существующие свойства нельзя будет удалить или настроить. Однако существующие свойства, доступные для записи, по-прежнему могут быть изменены.

После вызова `Object.seal()` объект нельзя будет вернуть в прежнее состояние. Чтобы определить, вызывался ли метод `Object.seal()` для объекта, можно вызвать метод `Object.isSealed()`.

Метод `Object.freeze()` обеспечивает еще более жесткую фиксацию объектов. Помимо того, что он делает объект нерасширяемым, а его свойства недоступными для настройки, он также делает все собственные свойства с данными доступными только для чтения. (Это не относится к свойствам объекта с методами доступа, обладающими методами записи; эти методы по-прежнему будут вызываться инструкциями присваивания.) Чтобы определить, вызывался ли метод `Object.freeze()` объекта, можно вызвать метод `Object.isFrozen()`.

Важно понимать, что `Object.seal()` и `Object.freeze()` воздействуют только на объект, который им передается: они не затрагивают прототип этого объекта. Если в программе потребуется полностью зафиксировать объект, вам, вероятно, потребуется зафиксировать также объекты в цепочке прототипов.

Все методы, `Object.preventExtensions()`, `Object.seal()` и `Object.freeze()`, возвращают переданный им объект, а это означает, что их можно использовать во вложенных вызовах:

```
// Создать нерасширяемый обьект с ненастраиваемыми свойствами, с жестко
// зафиксированным прототипом и свойством, недоступным для перечисления
var o = Object.seal(Object.create(Object.freeze({x:1}),
                                  {y: {value: 2, writable: true}}));
```

6.9. Сериализация объектов

Сериализация объектов – это процесс преобразования объектов в строковую форму представления, которая позднее может использоваться для их восстановления. Для сериализации и восстановления объектов JavaScript стандартом ECMAScript 5 предоставляются встроенные функции `JSON.stringify()` и `JSON.parse()`. Эти функции используют формат обмена данными JSON. Название JSON происходит от «JavaScript Object Notation» (форма записи объектов JavaScript), а синтаксис этой формы записи напоминает синтаксис литералов объектов и массивов в языке JavaScript:

```
o = {x:1, y:{z:[false,null,""]}}; // Определить испытательный обьект
s = JSON.stringify(o);            // s == '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);                // p - глубокая копия обьекта o
```

Базовые реализации этих функций в ECMAScript 5 очень точно повторяют общедоступные реализации в ECMAScript 3, доступные в *http://json.org/json2.js*. С практической точки зрения это совершенно одинаковые реализации, и эти функции стандарта ECMAScript 5 можно использовать в ECMAScript 3, подключив указанный выше модуль *json2.js*.

Синтаксис формата JSON является лишь *подмножеством* синтаксиса языка JavaScript и не может использоваться для представления всех возможных значений, допустимых в JavaScript. Поддерживаются и могут быть сериализованы и восстановлены: объекты, массивы, строки, конечные числовые значения, `true`, `false` и `null`. Значения `NaN`, `Infinity` и `-Infinity` сериализуются в значение `null`. Объекты `Date` сериализуются в строки с датами в формате ISO (смотрите описание функции `Date.toJSON()`), но `JSON.parse()` оставляет их в строковом представлении и не восстанавливает первоначальные объекты `Date`. Объекты `Function`, `RegExp` и `Error` и значение `undefined` не могут быть сериализованы или восстановлены. Функция `JSON.stringify()` сериализует только перечислимые собственные свойства объекта. Если значение свойства не может быть сериализовано, это свойство просто исключается из строкового представления. Обе функции, `JSON.stringify()` и `JSON.parse()`, принимают необязательный второй аргумент, который можно использовать для настройки процесса сериализации и/или восстановления, например, посредством определения списка свойств, подлежащих сериализации, или функции преобразования значений во время сериализации. В справочном разделе приводится полное описание этих функций.

6.10. Методы класса Object

Как описывалось выше, все объекты в языке JavaScript (за исключением тех, что явно созданы без прототипа) наследуют свойства от `Object.prototype`. Эти наследуемые свойства являются первичными методами и представляют особый интерес для программистов на JavaScript, потому что доступны повсеместно. Мы уже познакомились с методами `hasOwnProperty()`, `propertyIsEnumerable()` и `isPrototypeOf()`. (И мы уже охватили достаточно много статических функций, определяемых конструктором `Object`, таких как `Object.create()` и `Object.getPrototypeOf()`.) В этом разделе описывается несколько универсальных методов объектов, которые определены в `Object.prototype` и предназначены для переопределения в других, более специализированных классах.

6.10.1. Метод toString()

Метод `toString()` не требует аргументов; он возвращает строку, каким-либо образом представляющую значение объекта, для которого он вызывается. Интерпретатор JavaScript вызывает этот метод объекта во всех тех случаях, когда ему требуется преобразовать объект в строку. Например, это происходит, когда используется оператор + для конкатенации строки с объектом, или при передаче объекта методу, требующему строку.

Метод `toString()` по умолчанию не очень информативен (однако его удобно использовать для определения класса объекта, как было показано в разделе 6.8.2). Например, следующий фрагмент просто записывает в переменную `s` строку `"[object Object]"`:

```
var s = { x:1, y:1 }.toString( );
```

Этот метод по умолчанию не отображает особенно полезной информации, поэтому многие классы определяют собственные версии метода `toString()`. Например, когда массив преобразуется в строку, мы получаем список элементов массива, каждый из которых преобразуется в строку, а когда в строку преобразуется функция, мы получаем исходный программный код этой функции. Эти специализированные версии метода `toString()` описываются в справочном руководстве. Смотрите, например, описание методов `Array.toString()`, `Date.toString()` и `Function.toString()`.

В разделе 9.6.3 описывается, как можно переопределить метод `toString()` для своих собственных классов.

6.10.2. Метод toLocaleString()

В дополнение к методу `toString()` все объекты имеют метод `toLocaleString()`. Назначение последнего состоит в получении локализованного строкового представления объекта. По умолчанию метод `toLocaleString()`, определяемый классом `Object`, никакой локализации не выполняет; он просто вызывает метод `toString()` и возвращает полученное от него значение. Классы `Date` и `Number` определяют собственные версии метода `toLocaleString()`, возвращающие строковые представления чисел и дат в соответствии с региональными настройками. Класс `Array` определяет версию метода `toLocaleString()`, действующую подобно методу `toString()` за исключением того, что он форматирует элементы массива вызовом их метода `toLocaleString()`, а не `toString()`.

6.10.3. Метод toJSON()

В действительности `Object.prototype` не определяет метод `toJSON()`, но метод `JSON.stringify()` (раздел 6.9) пытается отыскать и использовать метод `toJSON()` любого объекта, который требуется сериализовать. Если объект обладает этим методом, он вызывается и сериализации подвергается возвращаемое значение, а не исходный объект. Примером может служить метод `Date.toJSON()`.

6.10.4. Метод valueOf()

Метод `valueOf()` во многом похож на метод `toString()`, но вызывается, когда интерпретатору JavaScript требуется преобразовать объект в значение какого-либо простого типа, отличного от строки, – обычно в число. Интерпретатор JavaScript вызывает этот метод автоматически, если объект используется в контексте значения простого типа. Метод `valueOf()` по умолчанию не выполняет ничего, что представляло бы интерес, но некоторые встроенные классы объектов переопределяют метод `valueOf()` (например, `Date.valueOf()`). В разделе 9.6.3 описывается, как можно переопределить метод `valueOf()` в собственных типах объектов.

7

Массивы

Массив – это упорядоченная коллекция значений. Значения в массиве называются *элементами*, и каждый элемент характеризуется числовой позицией в массиве, которая называется *индексом*. Массивы в языке JavaScript являются *нетипизированными*: элементы массива могут иметь любой тип, причем разные элементы одного и того же массива могут иметь разные типы. Элементы массива могут даже быть объектами или другими массивами, что позволяет создавать сложные структуры данных, такие как массивы объектов и массивы массивов. Отсчет индексов массивов в языке JavaScript начинается с нуля и для них используются 32-битные целые числа: первый элемент массива имеет индекс 0, а наибольший возможный индекс имеет значение 4294967294 (2^{32}-2), т. е. максимально возможный размер массива составляет 4294967295 элементов. Массивы в JavaScript являются *динамическими*: они могут увеличиваться и уменьшаться в размерах по мере необходимости; нет необходимости объявлять фиксированные размеры массивов при их создании или повторно распределять память при изменении их размеров. Массивы в JavaScript могут быть *разреженными*: не требуется, чтобы массив содержал элементы с непрерывной последовательностью индексов – в массивах могут отсутствовать элементы с некоторыми индексами. Все массивы в JavaScript имеют свойство `length`. Для неразреженных массивов это свойство определяет количество элементов в массиве. Для разреженных массивов значение `length` больше числа всех элементов в массиве.

Массивы в языке JavaScript – это специализированная форма объектов, а индексы массивов означают чуть больше, чем просто имена свойств, которые по совпадению являются целыми числами. Мы еще не раз будем говорить о специфических особенностях массивов повсюду в этой главе. Реализации обычно оптимизируют операции с массивами, благодаря чему доступ к элементам массивов по их числовым индексам выполняется значительно быстрее, чем доступ к обычным свойствам объектов.

Массивы наследуют свои свойства от прототипа `Array.prototype`, который определяет богатый набор методов манипулирования массивами, о которых рассказывается в разделах 7.8 и 7.9. Большинство из этих методов являются *универсаль-*

ными, т. е. они могут применяться не только к истинным массивам, но и к любым объектам, «похожим на массивы». Объекты, похожие на массивы, будут рассматриваться в разделе 7.11. В ECMAScript 5 строки ведут себя как массивы символов, и мы обсудим такое их поведение в разделе 7.12.

7.1. Создание массивов

Легче всего создать массив с помощью литерала, который представляет собой простой список разделенных запятыми элементов массива в квадратных скобках. Например:

```
var empty = [];                // Пустой массив
var primes = [2, 3, 5, 7, 11]; // Массив с пятью числовыми элементами
var misc = [ 1.1, true, "a", ]; // 3 элемента разных типов + завершающая запятая
```

Значения в литерале массива не обязательно должны быть константами – это могут быть любые выражения:

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

Литералы массивов могут содержать литералы объектов или литералы других массивов:

```
var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
```

Если литерал массива содержит несколько идущих подряд запятых без значений между ними, создается разреженный массив (подробнее об этом рассказывается в разделе 7.3). Элементы, соответствующие таким пропущенным значениям, отсутствуют в массиве, но при обращении к ним возвращается значение `undefined`:

```
var count = [1,,3]; // Элементы с индексами 0 и 2. count[1] => undefined
var undefs = [,,];  // Массив без элементов, но с длиной, равной 2
```

Синтаксис литералов массивов позволяет вставлять необязательную завершающую запятую, т. е. литерал `[,,]` соответствует массиву с двумя элементами, а не с тремя.

Другой способ создания массива состоит в вызове конструктора `Array()`. Вызвать конструктор можно тремя разными способами:

- Вызвать конструктор без аргументов:

  ```
  var a = new Array();
  ```

  В этом случае будет создан пустой массив, эквивалентный литералу `[]`.

- Вызвать конструктор с единственным числовым аргументом, определяющим длину массива:

  ```
  var a = new Array(10);
  ```

  В этом случае будет создан пустой массив указанной длины. Такая форма вызова конструктора `Array()` может использоваться для предварительного распределения памяти под массив, если заранее известно количество его элементов. Обратите внимание, что при этом в массиве не сохраняется никаких значений и даже свойства-индексы массива с именами «0», «1» и т. д. в массиве не определены.

- Явно указать в вызове конструктора значения первых двух или более элементов массива или один нечисловой элемент:

  ```
  var a = new Array(5, 4, 3, 2, 1, "testing, testing");
  ```

  В этом случае аргументы конструктора становятся значениями элементов нового массива. Использование литералов массивов практически всегда проще, чем подобное применение конструктора `Array()`.

7.2. Чтение и запись элементов массива

Доступ к элементам массива осуществляется с помощью оператора `[]`. Слева от скобок должна присутствовать ссылка на массив. Внутри скобок должно находиться произвольное выражение, возвращающее неотрицательное целое значение. Этот синтаксис пригоден как для чтения, так и для записи значения элемента массива. Следовательно, допустимы все приведенные далее JavaScript-инструкции:

```
var a = ["world"];  // Создать массив с одним элементом
var value = a[0];   // Прочитать элемент 0
a[1] = 3.14;        // Записать значение в элемент 1
i = 2;
a[i] = 3;           // Записать значение в элемент 2
a[i + 1] = "hello"; // Записать значение в элемент 3
a[a[i]] = a[0];     // Прочитать элементы 0 и 2, записать значение в элемент 3
```

Напомню, что массивы являются специализированной разновидностью объектов. Квадратные скобки, используемые для доступа к элементам массива, действуют точно так же, как квадратные скобки, используемые для доступа к свойствам объекта. Интерпретатор JavaScript преобразует указанные в скобках числовые индексы в строки – индекс `1` превращается в строку `"1"`, – а затем использует строки как имена свойств. В преобразовании числовых индексов в строки нет ничего особенного: то же самое можно проделывать с обычными объектами:

```
o = {};             // Создать простой объект
o[1] = "one";       // Индексировать его целыми числами
```

Особенность массивов состоит в том, что при использовании имен свойств, которые являются неотрицательными целыми числами, не превышающими 2^{32}-2, массивы автоматически определяют значение свойства `length`. Например, выше был создан массив `a` с единственным элементом. Затем были присвоены значения его элементам с индексами 1, 2 и 3. В результате этих операций значение свойства `length` массива изменилось:

```
a.length          // => 4
```

Следует четко отличать *индексы в массиве* от *имен свойств объектов*. Все индексы являются именами свойств, но только свойства с именами, представленными целыми числами в диапазоне от 0 до 2^{32}-2 являются индексами. Все массивы являются объектами, и вы можете добавлять к ним свойства с любыми именами. Однако если вы затрагиваете свойства, которые являются индексами массива, массивы реагируют на это, обновляя значение свойства `length` при необходимости.

Обратите внимание, что в качестве индексов массивов допускается использовать отрицательные и не целые числа. В этом случае числа преобразуются в строки, которые используются как имена свойств. Когда имя свойства не является неотрицательным целым числом, оно интерпретируется как имя обычного свойства объекта, а не как индекс массива. Кроме того, при индексировании массива строками, которые являются представлениями неотрицательных целых чисел, они интерпретируются как индексы массива, а не как свойства объекта. То же относится и к вещественным числам, не имеющим дробной части:

```
a[-1.23] = true;   // Будет создано свойство с именем "-1.23"
a["1000"] = 0;     // 1001-й элемент массива
a[1.000]           // Элемент с индексом 1. То же, что и a[1]
```

То обстоятельство, что индексы массива являются всего лишь особой разновидностью имен свойств объекта, означает, что для массивов JavaScript отсутствует понятие ошибки «выхода за границы». Попытка получить значение любого несуществующего свойства любого объекта не рассматривается как ошибка, в этом случае просто возвращается значение `undefined`. То же относится и к массивам:

```
a = [true, false]; // Этот массив имеет элементы с индексами 0 и 1
a[2]               // => undefined. Нет элемента с таким индексом.
a[-1]              // => undefined. Нет свойства с таким именем.
```

Поскольку массивы фактически являются объектами, они могут наследовать элементы от своих прототипов. В ECMAScript 5 массивы могут даже иметь элементы, определяющие методы чтения и записи (раздел 6.6). Если массив наследует элементы или элементы в нем имеют методы доступа, доступ к такому массиву не оптимизируется интерпретатором: время доступа к элементам такого массива будет сопоставимо с временем поиска обычных свойств объекта.

7.3. Разреженные массивы

Разреженным называется массив, индексы элементов которого не образуют непрерывную последовательность чисел, начиная с 0. Обычно свойство `length` массива определяет количество элементов в массиве. В разреженном массиве значение свойства `length` больше количества элементов. Разреженный массив можно создать с помощью конструктора `Array()` или путем присваивания значения элементу с индексом, большим, чем текущая длина массива.

```
a = new Array(5);  // Нет элементов, но a.length имеет значение 5.
a = [];            // Создаст пустой массив со значением length = 0.
a[1000] = 0;       // Добавит один элемент, но установит длину равной 1001.
```

Далее будет показано, что разреженный массив можно также создать с помощью оператора `delete`.

Существенно разреженные массивы обычно более медленны и потребляют больше памяти, чем плотные массивы, а поиск элементов в таких массивах занимает примерно столько же времени, что и поиск обычных свойств объектов.

Обратите внимание, что литералы с пропущенными значениями (когда в определении подряд следуют запятые, например `[1,,3]`) создают разреженные массивы, в которых пропущенные элементы просто не существуют:

```
var a1 = [,];            // Массив без элементов с длиной, равной 1
var a2 = [undefined];    // Массив с одним неопределенным элементом
0 in a1               // => false: a1 не имеет элемента с индексом 0
0 in a2               // => true: a2 имеет элемент с индексом 0 и со значением undefined
```

Некоторые старые реализации (такие как Firefox 3) некорректно вставляли элементы со значением `undefined` на место пропущенных элементов. В этих реализациях литерал `[1,,3]` был эквивалентен литералу `[1,undefined,3]`.

7.4. Длина массива

Любой массив имеет свойство `length`, и это свойство отличает массивы от обычных объектов JavaScript. Для плотных (т. е. неразреженных) массивов свойство `length` определяет количество элементов в массиве. Его значение на единицу больше самого большого индекса в массиве:

```
[].length             // => 0: массив не имеет элементов
['a','b','c'].length  // => 3: наибольший индекс равен 2, длина равна 3
```

Для разреженных массивов значение свойства `length` больше числа элементов, и все, что можно сказать в этом случае, – это то, что значение свойства `length` гарантированно будет превышать индекс любого элемента в массиве. Или, говоря иначе, массивы (разреженные или нет) никогда не будут содержать элемент, индекс которого будет больше или равен значению свойства `length` массива. Для поддержки этого свойства массивы проявляют две особенности поведения. Первая была описана выше: если присвоить значение элементу массива, индекс `i` которого больше или равен текущему значению свойства `length`, в свойство `length` записывается значение `i+1`.

Вторая особенность в поведении, обеспечивающем работу свойства `length`, заключается в том, что при присваивании свойству `length` неотрицательного целого числа `n`, меньшего, чем его текущее значение, все элементы массива с индексами, большими или равными значению `n`, удаляются из массива:

```
a = [1,2,3,4,5];      // Создать массив с пятью элементами.
a.length = 3;         // теперь массив a содержит элементы [1,2,3].
a.length = 0;         // Удалит все элементы. a - пустой массив [].
a.length = 5;         // Длина равна 5, но элементы отсутствуют, подобно Array(5)
```

В свойство `length` массива можно также записать значение больше, чем его текущее значение. В этом случае в массив не добавляются новые элементы, а просто создается разреженная область в конце массива.

В ECMAScript 5 свойство `length` массива можно сделать доступным только для чтения, с помощью `Object.defineProperty()` (раздел 6.7):

```
a = [1,2,3];                        // Создать массив a с тремя элементами.
Object.defineProperty(a, "length",  // Сделать свойство length
                   {writable: false}); // доступным только для чтения.
a.length = 0;                       // a не изменится.
```

Аналогично, если сделать элемент массива ненастраиваемым, его нельзя будет удалить. Если элемент нельзя будет удалить, то и свойство `length` не может быть установлено в значение, меньшее или равное индексу ненастраиваемого элемен-

та. (Смотрите раздел 6.7, а также описание методов `Object.seal()` и `Object.freeze()` в разделе 6.8.3.)

7.5. Добавление и удаление элементов массива

Мы уже видели, что самый простой способ добавить элементы в массив заключается в том, чтобы присвоить значения новым индексам:

```
a = []              // Создать пустой массив.
a[0] = "zero";      // И добавить элементы.
a[1] = "one";
```

Для добавления одного или более элементов в конец массива можно также использовать метод `push()`:

```
a = [];             // Создать пустой массив
a.push("zero")      // Добавить значение в конец. a = ["zero"]
a.push("one", "two") // Добавить еще два значения. a = ["zero", "one", "two"]
```

Добавить элемент в конец массива можно также, присвоив значение элементу `a[a.length]`. Для вставки элемента в начало массива можно использовать метод `unshift()` (описывается в разделе 7.8), при этом существующие элементы в массиве смещаются в позиции с более высокими индексами.

Удалять элементы массива можно с помощью оператора `delete`, как обычные свойства объектов:

```
a = [1,2,3];
delete a[1];        // теперь в массиве a отсутствует элемент с индексом 1
1 in a              // => false: индекс 1 в массиве не определен
a.length            // => 3: оператор delete не изменяет свойство length массива
```

Удаление элемента напоминает (но несколько отличается) присваивание значения `undefined` этому элементу. Обратите внимание, что применение оператора `delete` к элементу массива не изменяет значение свойства `length` и не сдвигает вниз элементы с более высокими индексами, чтобы заполнить пустоту, оставшуюся после удаления элемента. После удаления элемента массив превращается в разреженный массив.

Кроме того, как уже было показано выше, имеется возможность удалять элементы в конце массива простым присваиванием нового значения свойству `length`. Массивы имеют метод `pop()` (противоположный методу `push()`), который уменьшает длину массива на 1 и возвращает значение удаленного элемента. Также имеется метод `shift()` (противоположный методу `unshift()`), который удаляет элемент в начале массива. В отличие от оператора `delete`, метод `shift()` сдвигает все элементы вниз на позицию ниже их текущих индексов. Методы `pop()` и `shift()` описываются в разделе 7.8 и в справочном разделе.

Наконец существует многоцелевой метод `splice()`, позволяющий вставлять, удалять и замещать элементы массивов. Он изменяет значение свойства `length` и сдвигает элементы массива с более низкими или высокими индексами по мере необходимости. Подробности приводятся в разделе 7.8.

7.6. Обход элементов массива

Наиболее часто для обхода элементов массива используется цикл `for` (раздел 5.5.3):

```
var keys = Object.keys(o); // Получить массив имен свойств объекта o
var values = []           // Массив для сохранения значений свойств
for(var i = 0; i < keys.length; i++) { // Для каждого элемента в массиве
    var key = keys[i];                 // Получить имя свойства по индексу
    values[i] = o[key];                // Сохранить значение в массиве values
}
```

Во вложенных циклах и в других контекстах, когда скорость работы имеет критическое значение, иногда можно увидеть такой оптимизированный способ выполнения итераций по массиву, когда длина массива определяется только один раз, а не в каждой итерации:

```
for(var i = 0, len = keys.length; i < len; i++) {
    // тело цикла осталось без изменений
}
```

В примерах выше предполагается, что выполняется обход плотного массива и все элементы содержат допустимые значения. В противном случае необходимо организовать проверку значений элементов массива перед их использованием. Если желательно исключить из обработки значения `null`, `undefined` и несуществующие элементы, проверку можно записать так:

```
for(var i = 0; i < a.length; i++) {
    if (!a[i]) continue; // Пропустить null, undefined и несуществ. элементы
    // тело цикла
}
```

Если необходимо пропустить только значение `undefined` и несуществующие элементы, проверку можно записать так:

```
for(var i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue; // Пропустить undefined + несуществ. эл.
    // тело цикла
}
```

Наконец, если необходимо пропустить только несуществующие элементы, а элементы со значением `undefined` обрабатывать как обычные элементы, проверку можно записать так:

```
for(var i = 0; i < a.length; i++) {
    if (!(i in a)) continue ; // Пропустить несуществующие элементы
    // тело цикла
}
```

Для обхода разреженных массивов можно также использовать цикл `for/in` (раздел 5.5.4). Этот цикл присваивает имена перечислимых свойств (включая индексы массива) переменной цикла. Отсутствующие индексы в итерациях не участвуют:

```
for(var index in sparseArray) {
    var value = sparseArray[index];
    // Далее следуют операции с индексами и значениями
}
```

Как отмечалось в разделе 6.5, цикл `for/in` может возвращать имена унаследованных свойств, такие как имена методов, добавленных в `Array.prototype`. По этой причине не следует использовать цикл `for/in` для обхода массивов, не предусмотрев дополнительной проверки для фильтрации нежелательных свойств. Для этого можно было бы использовать, например, такие проверки:

```
for(var i in a) {
    if (!a.hasOwnProperty(i)) continue; // Пропустить унаследованные свойства
    // тело цикла
}

for(var i in a) {
    // Пропустить i, если оно не является целым неотрицательным числом
    if (String(Math.floor(Math.abs(Number(i)))) !== i) continue;
}
```

Спецификация ECMAScript допускает возможность обхода свойств объекта в цикле `for/in` в любом порядке. Обычно реализации обеспечивают обход индексов массивов в порядке возрастания, но это не гарантируется. В частности, если массив имеет и свойства объекта, и элементы массива, имена свойств могут возвращаться в порядке их создания, а не в порядке возрастания числовых значений. Разные реализации по-разному обрабатывают эту ситуацию, поэтому, если для вашего алгоритма порядок выполнения итераций имеет значение, вместо цикла `for/in` лучше использовать обычный цикл `for`.

Стандарт ECMAScript 5 определяет множество новых методов, позволяющих выполнять итерации по элементам массивов в порядке возрастания индексов и передавать их функции, определяемой пользователем. Наиболее типичным представителем этих методов является метод `forEach()`:

```
var data = [1,2,3,4,5];     // Этот массив требуется обойти
var sumOfSquares = 0;       // Требуется вычислить сумму квадратов элементов
data.forEach(function(x) {  // Передать каждый элемент этой функции
                sumOfSquares += x*x; // прибавить квадрат к сумме
            });
sumOfSquares                // =>55 : 1+4+9+16+25
```

`forEach()` и другие родственные методы, предназначенные для выполнения итераций, позволяют использовать при работе с массивами простой и мощный стиль функционального программирования. Они описываются в разделе 7.9, и еще раз мы вернемся к ним в разделе 8.8, когда будем рассматривать приемы функционального программирования.

7.7. Многомерные массивы

JavaScript не поддерживает «настоящие» многомерные массивы, но позволяет неплохо имитировать их при помощи массивов из массивов. Для доступа к элементу данных в массиве массивов достаточно дважды использовать оператор `[]`. Например, предположим, что переменная `matrix` – это массив массивов чисел. Каждый элемент `matrix[x]` – это массив чисел. Для доступа к определенному числу в массиве можно использовать выражение `matrix[x][y]`. Ниже приводится конкретный пример, где двумерный массив используется в качестве таблицы умножения:

```
// Создать многомерный массив
var table = new Array(10);          // В таблице 10 строк
for(var i = 0; i < table.length; i++)
    table[i] = new Array(10);       // В каждой строке 10 столбцов

// Инициализировать массив
for(var row = 0; row < table.length; row++) {
    for(col = 0; col < table[row].length; col++) {
        table[row][col] = row*col;
    }
}

// Расчет произведения 5*7 с помощью многомерного массива
var product = table[5][7];          // 35
```

7.8. Методы класса Array

Стандарт ECMAScript 3 определяет в составе `Array.prototype` **множество удобных функций для работы с массивами, которые доступны как методы любого массива. Эти методы будут представлены в следующих подразделах. Более полную информацию можно найти в разделе** `Array` **в справочной части по базовому языку JavaScript. Стандарт ECMAScript 5 определяет дополнительные методы для выполнения итераций по массивам – эти методы рассматриваются в разделе 7.9.**

7.8.1. Метод join()

Метод `Array.join()` **преобразует все элементы массива в строки, объединяет их и возвращает получившуюся строку. В необязательном аргументе методу можно передать строку, которая будет использоваться для отделения элементов в строке результата. Если строка-разделитель не указана, используется запятая. Например, следующий фрагмент дает в результате строку «1,2,3»:**

```
var a = [1, 2, 3];      // Создать новый массив с указанными тремя элементами
a.join();               // => "1,2,3"
a.join(" ");            // => "1 2 3"
a.join("");             // => "123"
var b = new Array(10);  // Массив с длиной, равной 10, и без элементов
b.join('-')             // => '---------': строка из 9 дефисов
```

Метод `Array.join()` **является обратным по отношению к методу** `String.split()`**, создающему массив путем разбиения строки на фрагменты.**

7.8.2. Метод reverse()

Метод `Array.reverse()` **меняет порядок следования элементов в массиве на обратный и возвращает переупорядоченный массив. Перестановка выполняется непосредственно в исходном массиве, т. е. этот метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве. Например, следующий фрагмент, где используются методы** `reverse()` **и** `join()`**, дает в результате строку** `"3,2,1"`**:**

```
var a = [1,2,3];
a.reverse().join(); // => "3,2,1": теперь a = [3,2,1]
```

7.8.3. Метод sort()

Метод `Array.sort()` сортирует элементы в исходном массиве и возвращает отсортированный массив. Если метод `sort()` вызывается без аргументов, сортировка выполняется в алфавитном порядке (для сравнения элементы временно преобразуются в строки, если это необходимо):

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", "); // s == "apple, banana, cherry"
```

Неопределенные элементы переносятся в конец массива.

Для сортировки в каком-либо ином порядке, отличном от алфавитного, методу `sort()` можно передать функцию сравнения в качестве аргумента. Эта функция устанавливает, какой из двух ее аргументов должен следовать раньше в отсортированном списке. Если первый аргумент должен предшествовать второму, функция сравнения должна возвращать отрицательное число. Если первый аргумент должен следовать за вторым в отсортированном массиве, то функция должна возвращать число больше нуля. А если два значения эквивалентны (т. е. порядок их следования не важен), функция сравнения должна возвращать 0. Поэтому, например, для сортировки элементов массива в числовом порядке можно сделать следующее:

```
var a = [33, 4, 1111, 222];
a.sort();              // Алфавитный порядок: 1111, 222, 33, 4
a.sort(function(a,b) { // Числовой порядок: 4, 33, 222, 1111
    return a-b;        // Возвращает значение < 0, 0 или > 0
    });                // в зависимости от порядка сортировки a и b
a.sort(function(a,b) {return b-a}); // Обратный числовой порядок
```

Обратите внимание, насколько удобно использовать в этом фрагменте неименованную функцию. Функция сравнения используется только здесь, поэтому нет необходимости давать ей имя.

В качестве еще одного примера сортировки элементов массива можно реализовать сортировку массива строк без учета регистра символов, передав функцию сравнения, преобразующую свои аргументы в нижний регистр (с помощью метода `toLowerCase()`) перед сравнением.

```
a = ['ant', 'Bug', 'cat', 'Dog']
a.sort(); // сортировка с учетом регистра символов: ['Bug','Dog','ant',cat']
a.sort(function(s,t) { // Сортировка без учета регистра символов
            var a = s.toLowerCase();
            var b = t.toLowerCase();
            if (a < b) return -1;
            if (a > b) return 1;
            return 0;
        });          // => ['ant','Bug','cat','Dog']
```

7.8.4. Метод concat()

Метод `Array.concat()` создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод `concat()`, и значения всех аргументов, переданных методу `concat()`. Если какой-либо из этих аргументов сам является массивом, его элементы добавляются в возвращаемый массив. Следует, однако, отметить, что рекурсивного превращения массива из массивов в одномерный массив не происходит. Метод concat() не изменяет исходный массив. Ниже приводится несколько примеров:

```
var a = [1,2,3];
a.concat(4, 5)         // Вернет [1,2,3,4,5]
a.concat([4,5]);       // Вернет [1,2,3,4,5]
a.concat([4,5],[6,7])  // Вернет [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]]) // Вернет [1,2,3,4,5,[6,7]]
```

7.8.5. Метод slice()

Метод Array.slice() возвращает *фрагмент*, или подмассив, указанного массива. Два аргумента метода определяют начало и конец возвращаемого фрагмента. Возвращаемый массив содержит элемент, номер которого указан в первом аргументе, плюс все последующие элементы, вплоть до (но не включая) элемента, номер которого указан во втором аргументе. Если указан только один аргумент, возвращаемый массив содержит все элементы от начальной позиции до конца массива. Если какой-либо из аргументов имеет отрицательное значение, он определяет номер элемента относительно конца массива. Так, аргументу -1 соответствует последний элемент массива, а аргументу -3 – третий элемент массива с конца. Вот несколько примеров:

```
var a = [1,2,3,4,5];
a.slice(0,3);   // Вернет [1,2,3]
a.slice(3);     // Вернет [4,5]
a.slice(1,-1);  // Вернет [2,3,4]
a.slice(-3,-2); // Вернет [3]
```

7.8.6. Метод splice()

Метод Array.splice() – это универсальный метод, выполняющий вставку или удаление элементов массива. В отличие от методов slice() и concat(), метод splice() изменяет исходный массив, относительно которого он был вызван. Обратите внимание, что методы splice() и slice() имеют очень похожие имена, но выполняют совершенно разные операции.

Метод splice() может удалять элементы из массива, вставлять новые элементы или выполнять обе операции одновременно. Элементы массива при необходимости смещаются, чтобы после вставки или удаления образовывалась непрерывная последовательность. Первый аргумент метода splice() определяет позицию в массиве, начиная с которой будет выполняться вставка и/или удаление. Второй аргумент определяет количество элементов, которые должны быть удалены (вырезаны) из массива. Если второй аргумент опущен, удаляются все элементы массива от указанного до конца массива. Метод splice() возвращает массив удаленных элементов или (если ни один из элементов не был удален) пустой массив. Например:

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);   // Вернет [5,6,7,8]; a = [1,2,3,4]
a.splice(1,2); // Вернет [2,3]; a = [1,4]
a.splice(1,1); // Вернет [4]; a = [1]
```

Первые два аргумента метода splice() определяют элементы массива, подлежащие удалению. За этими аргументами может следовать любое количество дополнительных аргументов, определяющих элементы, которые будут вставлены в массив, начиная с позиции, указанной в первом аргументе. Например:

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b'); // Вернет []; a = [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3); // Вернет ['a','b']; a = [1,2,[1,2],3,3,4,5]
```

Обратите внимание, что, в отличие от `concat()`, метод `splice()` вставляет массивы целиком, а не их элементы.

7.8.7. Методы push() и pop()

Методы `push()` и `pop()` позволяют работать с массивами как со стеками. Метод `push()` добавляет один или несколько новых элементов в конец массива и возвращает его новую длину. Метод `pop()` выполняет обратную операцию – удаляет последний элемент массива, уменьшает длину массива и возвращает удаленное им значение. Обратите внимание, что оба эти метода изменяют исходный массив, а не создают его модифицированную копию. Комбинация `push()` и `pop()` позволяет на основе массива реализовать стек с дисциплиной обслуживания «первым вошел – последним вышел». Например:

```
var stack = [];         // стек: []
stack.push(1,2);        // стек: [1,2]     Вернет 2
stack.pop();            // стек: [1]       Вернет 2
stack.push(3);          // стек: [1,3]     Вернет 2
stack.pop();            // стек: [1]       Вернет 3
stack.push([4,5]);      // стек: [1,[4,5]] Вернет 2
stack.pop()             // стек: [1]       Вернет [4,5]
stack.pop();            // стек: []        Вернет 1
```

7.8.8. Методы unshift() и shift()

Методы `unshift()` и `shift()` ведут себя почти так же, как `push()` и `pop()`, за исключением того, что они вставляют и удаляют элементы в начале массива, а не в конце. Метод `unshift()` смещает существующие элементы в сторону больших индексов для освобождения места, добавляет элемент или элементы в начало массива и возвращает новую длину массива. Метод `shift()` удаляет и возвращает первый элемент массива, смещая все последующие элементы на одну позицию вниз, чтобы занять место, освободившееся в начале массива. Например:

```
var a = [];             // a:[]
a.unshift(1);           // a:[1]         Вернет: 1
a.unshift(22);          // a:[22,1]      Вернет: 2
a.shift();              // a:[1]         Вернет: 22
a.unshift(3,[4,5]);     // a:[3,[4,5],1] Вернет: 3
a.shift();              // a:[[4,5],1]   Вернет: 3
a.shift();              // a:[1]         Вернет: [4,5]
a.shift();              // a:[]          Вернет: 1
```

Обратите внимание на поведение метода `unshift()` при вызове с несколькими аргументами. Аргументы вставляются не по одному, а все сразу (как в случае с методом `splice()`). Это значит, что в результирующем массиве они будут следовать в том же порядке, в котором были указаны в списке аргументов. Будучи вставленными по одному, они бы расположились в обратном порядке.

7.8.9. Методы toString() и toLocaleString()

Массивы, как и любые другие объекты в JavaScript, имеют метод `toString()`. Для массива этот метод преобразует каждый его элемент в строку (вызывая в случае необходимости методы `toString()` элементов массива) и выводит список этих строк через запятую. Примечательно, что результат не включает квадратные скобки или какие-либо другие разделители вокруг значений массива. Например:

```
[1,2,3].toString()          // Получается '1,2,3'
["a", "b", "c"].toString() // Получается 'a,b,c'
[1, [2,'c']].toString()     // Получается '1,2,c'
```

Обратите внимание, что `toString()` возвращает ту же строку, что и метод `join()` при вызове его без аргументов.

Метод `toLocaleString()` – это локализованная версия `toString()`. Каждый элемент массива преобразуется в строку вызовом метода `toLocaleString()` элемента, а затем полученные строки объединяются с использованием специфического для региона (и определяемого реализацией) разделителя.

7.9. Методы класса Array, определяемые стандартом ECMAScript 5

Стандарт ECMAScript 5 определяет девять новых методов массивов, позволяющих выполнять итерации, отображение, фильтрацию, проверку, свертку и поиск. Все эти методы описываются в следующих далее подразделах.

Однако, прежде чем перейти к изучению особенностей, следует сделать некоторые обобщения, касающиеся методов массивов в ECMAScript 5. Во-первых, большинство описываемых ниже методов принимают функцию в первом аргументе и вызывают ее для каждого элемента (или нескольких элементов) массива. В случае разреженных массивов указанная функция не будет вызываться для несуществующих элементов. В большинстве случаев указанной функции передаются три аргумента: значение элемента массива, индекс элемента и сам массив. Чаще всего вам необходим будет только первый аргумент, а второй и третий аргументы можно просто игнорировать. Большинство методов массивов, введенных стандартом ECMAScript 5, которые в первом аргументе принимают функцию, также принимают второй необязательный аргумент. Если он указан, функция будет вызываться, как если бы она была методом этого второго аргумента. То есть второй аргумент будет доступен функции, как значение ключевого слова `this`. Значение, возвращаемое функцией, играет важную роль, но разные методы обрабатывают его по-разному. Ни один из методов массивов, введенных стандартом ECMAScript 5, не изменяет исходный массив. Разумеется, функция, передаваемая этим методам, может модифицировать исходный массив.

7.9.1. Метод forEach()

Метод `forEach()` выполняет обход элементов массива и для каждого из них вызывает указанную функцию. Как уже говорилось выше, функция передается методу `forEach()` в первом аргументе. При вызове этой функции метод `forEach()` будет передавать ей три аргумента: значение элемента массива, индекс элемента и сам массив. Если вас интересует только значение элемента, можно написать функцию с одним параметром – дополнительные аргументы будут игнорироваться:

```
var data = [1,2,3,4,5]; // Массив, элементы которого будут суммироваться
// Найти сумму элементов массива
var sum = 0;             // Начальное значение суммы 0
data.forEach(function(value) { sum += value; }); // Прибавить значение к sum
sum                      // => 15

// Увеличить все элементы массива на 1
data.forEach(function(v, i, a) { a[i] = v + 1; });
data                     // => [2,3,4,5,6]
```

Обратите внимание, что метод `forEach()` не позволяет прервать итерации, пока все элементы не будут переданы функции. То есть отсутствует эквивалент инструкции `break`, которую можно использовать с обычным циклом `for`. Если потребуется прервать итерации раньше, внутри функции можно возбуждать исключение, а вызов `forEach()` помещать в блок `try`. Ниже демонстрируется функция `foreach()`, вызывающая метод `forEach()` внутри такого блока `try`. Если функция, которая передается функции `foreach()`, возбудит исключение `foreach.break`, цикл будет прерван преждевременно:

```
function foreach(a,f,t) {
    try { a.forEach(f,t); }
    catch(e) {
        if (e === foreach.break) return;
        else throw e;
    }
}
foreach.break = new Error("StopIteration");
```

7.9.2. Метод map()

Метод `map()` передает указанной функции каждый элемент массива, относительно которого он вызван, и возвращает массив значений, возвращаемых этой функцией. Например:

```
a = [1, 2, 3];
b = a.map(function(x) { return x*x; }); // b = [1, 4, 9]
```

Метод `map()` вызывает функцию точно так же, как и метод `forEach()`. Однако функция, передаваемая методу `map()`, должна возвращать значение. Обратите внимание, что `map()` возвращает новый массив: он не изменяет исходный массив. Если исходный массив является разреженным, возвращаемый массив также будет разреженным: он будет иметь ту же самую длину и те же самые отсутствующие элементы.

7.9.3. Метод filter()

Метод `filter()` возвращает массив, содержащий подмножество элементов исходного массива. Передаваемая ему функция должна быть функцией-предикатом, т. е. должна возвращать значение `true` или `false`. Метод `filter()` вызывает функцию точно так же, как методы `forEach()` и `map()`. Если возвращается `true` или значение, которое может быть преобразовано в `true`, переданный функции элемент считается членом подмножества и добавляется в массив, возвращаемый методом. Например:

```
a = [5, 4, 3, 2, 1];
smallvalues = a.filter(function(x) { return x < 3 });   // [2, 1]
everyother = a.filter(function(x,i) { return i%2==0 }); // [5, 3, 1]
```

Обратите внимание, что метод `filter()` пропускает отсутствующие элементы в разреженных массивах и всегда возвращает плотные массивы. Чтобы уплотнить разреженный массив, можно выполнить следующие действия:

```
var dense = sparse.filter(function() { return true; });
```

А чтобы уплотнить массив и удалить из него все элементы со значениями `undefined` и `null`, можно использовать метод `filter()`, как показано ниже:

```
a = a.filter(function(x) { return x !== undefined && x != null; });
```

7.9.4. Методы every() и some()

Методы `every()` и `some()` являются предикатами массива: они применяют указанную функцию-предикат к элементам массива и возвращают `true` или `false`. Метод `every()` напоминает математический квантор всеобщности ∀: он возвращает `true`, только если переданная вами функция-предикат вернула `true` для всех элементов массива:

```
a = [1,2,3,4,5];
a.every(function(x) { return x < 10; })      // => true: все значения < 10.
a.every(function(x) { return x % 2 === 0; }) // => false: не все четные.
```

Метод `some()` напоминает математический квантор существования ∃: он возвращает `true`, если в массиве имеется хотя бы один элемент, для которого функция-предикат вернет `true`, а значение `false` возвращается методом, только если функция-предикат вернет `false` для всех элементов массива:

```
a = [1,2,3,4,5];
a.some(function(x) { return x%2===0; }) // => true: имеются четные числа.
a.some(isNaN)                           // => false: нет нечисловых элементов.
```

Обратите внимание, что оба метода, `every()` и `some()`, прекращают обход элементов массива, как только результат становится известен. Метод `some()` возвращает `true`, как только функция-предикат вернет `true`, и выполняет обход всех элементов массива, только если функция-предикат всегда возвращает `false`. Метод `every()` является полной противоположностью: он возвращает `false`, как только функция-предикат вернет `false`, и выполняет обход всех элементов массива, только если функция-предикат всегда возвращает `true`. Кроме того, отметьте, что в соответствии с правилами математики для пустого массива метод `every()` возвращает `true`, а метод `some()` возвращает `false`.

7.9.5. Методы reduce() и reduceRight()

Методы `reduce()` и `reduceRight()` объединяют элементы массива, используя указанную вами функцию, и возвращают единственное значение. Это типичная операция в функциональном программировании, где она известна также под названием «свертка». Примеры ниже помогут понять суть этой операции:

```
var a = [1,2,3,4,5]
var sum = a.reduce(function(x,y) { return x+y }, 0);     // Сумма значений
var product = a.reduce(function(x,y) { return x*y }, 1); // Произвед. значений
var max = a.reduce(function(x,y) { return (x>y)?x:y; }); // Наибольш. значение
```

Метод `reduce()` принимает два аргумента. В первом передается функция, которая выполняет операцию свертки. Задача этой функции – объединить некоторым способом или свернуть два значения в одно и вернуть свернутое значение. В примерах выше функции выполняют объединение двух значений, складывая их, умножая и выбирая наибольшее. Во втором (необязательном) аргументе передается начальное значение для функции.

Функции, передаваемые методу `reduce()`, отличаются от функций, передаваемых методам `forEach()` и `map()`. Знакомые уже значение, индекс и массив передаются им во втором, третьем и четвертом аргументах. А в первом аргументе передается накопленный результат свертки. При первом вызове в первом аргументе функции передается начальное значение, переданное методу `reduce()` во втором аргументе. Во всех последующих вызовах передается значение, полученное в результате предыдущего вызова функции. В первом примере, из приведенных выше, функция свертки сначала будет вызвана с аргументами 0 и 1. Она сложит эти числа и вернет 1. Затем она будет вызвана с аргументами 1 и 2 и вернет 3. Затем она вычислит 3+3=6, затем 6+4=10 и, наконец, 10+5=15. Это последнее значение 15 будет возвращено методом `reduce()`.

Возможно, вы обратили внимание, что в третьем вызове, в примере выше, методу `reduce()` передается единственный аргумент: здесь не указано начальное значение. Когда метод `reduce()` вызывается без начального значения, как в данном случае, в качестве начального значения используется первый элемент массива. Это означает, что при первом вызове функции свертки будут переданы первый и второй элементы массива. В примерах вычисления суммы и произведения точно так же можно было бы опустить аргумент с начальным значением.

Вызов метода `reduce()` с пустым массивом без начального значения вызывает исключение `TypeError`. Если вызвать метод с единственным значением – с массивом, содержащим единственный элемент, и без начального значения или с пустым массивом и начальным значением – он просто вернет это единственное значение, не вызывая функцию свертки.

Метод `reduceRight()` действует точно так же, как и метод `reduce()`, за исключением того, что массив обрабатывается в обратном порядке, от больших индексов к меньшим (справа налево). Это может потребоваться, если операция свертки имеет ассоциативность справа налево, например:

```
var a = [2, 3, 4]
// Вычислить 2^(3^4). Операция возведения в степень имеет ассоциативность справа налево
var big = a.reduceRight(function(accumulator,value) {
                          return Math.pow(value,accumulator);
                       });
```

Обратите внимание, что ни `reduce()`, ни `reduceRight()` не принимают необязательный аргумент, определяющий значение `this` внутри функции свертки. Его место занял необязательный аргумент с начальным значением. Если потребуется вызывать функцию свертки как метод конкретного объекта, можно воспользоваться методом `Function.bind()`.

Следует отметить, что методы `every()` и `some()`, описанные выше, являются своеобразной разновидностью операции свертки массива. Однако они отличаются от `reduce()` тем, что стремятся завершить обход массива как можно раньше и не всегда проверяют значения всех его элементов.

В примерах, представленных до сих пор, для простоты использовались числовые массивы, но методы `reduce()` и `reduceRight()` могут использоваться не только для математических вычислений. Взгляните на функцию `union()` в примере 6.2. Она вычисляет «объединение» двух объектов и возвращает новый объект, имеющий свойства обоих. Эта функция принимает два объекта и возвращает другой объект, т. е. она действует как функция свертки, поэтому ее можно использовать с методом `reduce()` и обобщить операцию создания объединения произвольного числа объектов:

```
var objects = [{x:1}, {y:2}, {z:3}];
var merged = objects.reduce(union); // => {x:1, y:2, z:3}
```

Напомню, что, когда два объекта имеют свойства с одинаковыми именами, функция `union()` использует значение свойства второго аргумента, т. е. `reduce()` и `reduceRight()` могут давать разные результаты при использовании с функцией `union()`:

```
var objects = [{x:1,a:1}, {y:2,a:2}, {z:3,a:3}];
var leftunion = objects.reduce(union);       // {x:1, y:2, z:3, a:3}
var rightunion = objects.reduceRight(union); // {x:1, y:2, z:3, a:1}
```

7.9.6. Методы indexOf() и lastIndexOf()

Методы `indexOf()` и `lastIndexOf()` отыскивают в массиве элемент с указанным значением и возвращают индекс первого найденного элемента или -1, если элемент с таким значением отсутствует. Метод `indexOf()` выполняет поиск от начала массива к концу, а метод `lastIndexOf()` – от конца к началу.

```
a = [0,1,2,1,0];
a.indexOf(1)       // => 1: a[1] = 1
a.lastIndexOf(1)   // => 3: a[3] = 1
a.indexOf(3)       // => -1: нет элемента со значением 3
```

В отличие от других методов, описанных в этом разделе, методы `indexOf()` и `lastIndexOf()` не принимают функцию в виде аргумента. В первом аргументе им передается искомое значение. Второй аргумент является необязательным: он определяет индекс массива, с которого следует начинать поиск. Если опустить этот аргумент, метод `indexOf()` начнет поиск с начала массива, а метод `lastIndexOf()` – с конца. Во втором аргументе допускается передавать отрицательные значения, которые интерпретируются как смещение относительно конца массива, как в методе `splice()`: значение -1, например, соответствует последнему элементу массива.

Следующая функция отыскивает заданное значение в массиве и возвращает массив всех индексов, где было найдено совпадение. Здесь демонстрируется, как можно использовать второй аргумент метода `indexOf()` для поиска совпадений после первого.

```
// Отыскивает все вхождения значения x в массив и возвращает
// массив индексов найденных совпадений
function findall(a, x) {
    var results = [],          // Возвращаемый массив индексов
        len = a.length,        // Длина массива, где выполняется поиск
        pos = 0;               // Начальная позиция поиска
    while(pos < len) {         // Пока остались непроверенные элементы...
        pos = a.indexOf(x, pos); // Искать
        if (pos === -1) break;  // Если ничего не найдено, поиск завершен.
        results.push(pos);     // Иначе - сохранить индекс в массиве
        pos = pos + 1;         // И продолжить поиск со следующего элемента
    }
    return results;            // Вернуть массив индексов
}
```

Обратите внимание, что строки также имеют методы `indexOf()` и `lastIndexOf()`, которые действуют подобно методам массивов.

7.10. Тип Array

На протяжении этой главы мы не раз имели возможность убедиться, что массивы являются объектами, обладающими особыми чертами поведения. Получая неизвестный объект, иногда бывает полезно проверить, является он массивом или нет. Сделать это в реализации ECMAScript 5 можно с помощью функции `Array.isArray()`:

```
Array.isArray([]) // => true
Array.isArray({}) // => false
```

Однако до выхода стандарта ECMAScript 5 отличить массивы от других объектов было удивительно сложно. Оператор `typeof` никак не помогает в этом: для массивов он возвращает строку «object» (и для всех других объектов, кроме функций). В простых случаях можно использовать оператор `instanceof`:

```
[] instanceof Array   // => true
({}) instanceof Array // => false
```

Проблема применения оператора `instanceof` состоит в том, что в веб-броузерах может быть открыто несколько окон или фреймов. Каждое окно или фрейм имеет собственное окружение JavaScript, с собственным глобальным объектом. А каждый глобальный объект имеет собственное множество функций-конструкторов. Поэтому объект из одного фрейма никогда не будет определяться как экземпляр конструктора в другом фрейме. Даже при том, что путаница между фреймами возникает довольно редко, тем не менее этого вполне достаточно, чтобы считать оператор `instanceof` ненадежным средством определения принадлежности к массивам.

Решение заключается в том, чтобы выполнить проверку атрибута `class` (раздел 6.8.2) объекта. Для массивов этот атрибут всегда будет иметь значение «Array», благодаря чему в реализации ECMAScript 3 функцию `isArray()` можно определить так:

```
var isArray = Function.isArray || function(o) {
    return typeof o === "object" &&
           Object.prototype.toString.call(o) === "[object Array]";
};
```

Фактически именно такая проверка атрибута `class` выполняется в функции `Array.isArray()`, определяемой стандартом ECMAScript 5. Прием определения класса объекта с помощью `Object.prototype.toString()` был описан в разделе 6.8.2 и продемонстрирован в примере 6.4.

7.11. Объекты, подобные массивам

Как мы уже видели, массивы в языке JavaScript обладают некоторыми особенностями, отсутствующими в других объектах:

- Добавление нового элемента вызывает автоматическое обновление свойства `length`.
- Уменьшение значения свойства `length` вызывает усечение массива.
- Массивы наследуют множество удобных методов от `Array.prototype`.
- Атрибут `class` массивов имеет значение «Array».

Все эти характеристики отличают массивы в языке JavaScript от других объектов. Но они не главное, что определяет массив. Часто бывает удобно организовать работу с произвольным объектом, как со своего рода массивом – через свойство `length` и соответствующие неотрицательные целочисленные свойства.

Такие объекты, «подобные массивам», иногда используются для решения практических задач, и хотя с ними нельзя работать через методы массивов или ожидать специфического поведения свойства `length`, все же можно организовать перебор свойств объекта теми же программными конструкциями, которые используются при работе с настоящими массивами. Оказывается, что значительное число алгоритмов для работы с массивами вполне пригодно для работы с объектами, подобными массивам. Это особенно верно, если используемые алгоритмы не изменяют массивы или хотя бы не затрагивают его свойство `length`.

В следующем фрагменте создается обычный объект и к нему добавляются дополнительные свойства, которые превращают его в объект, подобный массиву, после чего производится перебор «элементов» получившегося псевдомассива:

```
var a = {}; // Для начала создать обычный пустой объект

// Добавить свойства, которые сделают его похожим на массив
var i = 0;
while(i < 10) {
    a[i] = i * i;
    i++;
}
a.length = i;

// Теперь можно обойти свойства объекта, как если бы он был настоящим массивом
var total = 0;
for(var j = 0; j < a.length; j++)
    total += a[j];
```

Объект `Arguments`, который описывается в разделе 8.3.2, является объектом, подобным массиву. В клиентском языке JavaScript такие объекты возвращаются многими методами объектной модели документа (DOM), такими как метод `document.getElementsByTagName()`. Следующая функция проверяет, является ли объект подобным массиву:

```
// Определяет, является ли о объектом, подобным массиву. Строки и функции имеют
// числовое свойство length, но они исключаются проверкой typeof.
// В клиентском JavaScript текстовые узлы DOM имеют числовое свойство length
// и, возможно, должны быть исключены дополнительной проверкой o.nodeType != 3.
function isArrayLike(o) {
    if (o &&                                // o не null, не undefined и т.д.
        typeof o === "object" &&            // o - объект
        isFinite(o.length) &&               // o.length - конечное число
        o.length >= 0 &&                    // o.length - положительное
        o.length===Math.floor(o.length) &&  // o.length - целое
        o.length < 4294967296)              // o.length < 2^32

        return true;                        // Значит, объект o подобен массиву
    else
        return false;                       // Иначе - нет
}
```

В разделе 7.12 будет показано, что строки в ECMAScript 5 ведут себя подобно массивам (и некоторые броузеры обеспечивали возможность обращения к символам в строке по индексам еще до выхода ECMAScript 5). Однако проверки на подобие массивам, такие как приведенная выше, для строк обычно возвращают `false` – с ними лучше работать как со строками, чем как с массивами.

Методы массивов в языке JavaScript преднамеренно были сделаны достаточно универсальными, чтобы их можно было использовать не только с настоящими массивами, но и с объектами, подобными массивам. В ECMAScript 5 все методы массивов являются универсальными. В ECMAScript 3 универсальными также являются все методы, за исключением `toString()` и `toLocaleString()`. (К исключениям также относится метод `concat()`: несмотря на то что его можно применять к объектам, подобным массивам, он некорректно разворачивает объекты в возвращаемый массив.) Поскольку объекты, подобные массивам, не наследуют свойства от `Array.prototype`, к ним нельзя напрямую применить методы массивов. Однако их можно вызывать косвенно, с помощью метода `Function.call()`:

```
var a = {"0":"a", "1":"b", "2":"c", length:3}; // Объект, подобный массиву
Array.prototype.join.call(a, "+")              // => "a+b+c"
Array.prototype.slice.call(a, 0) // => ["a","b","c"]: копия, настоящий массив
Array.prototype.map.call(a, function(x) {
    return x.toUpperCase();
})                                             // => ["A","B","C"]:
```

Мы уже встречались с таким использованием метода `call()` в разделе 7.10, где описывался метод `isArray()`. Метод `call()` объектов класса `Function` детально рассматривается в разделе 8.7.3.

Методы массивов, определяемые в ECMAScript 5, были введены в Firefox 1.5. Поскольку они имели универсальную реализацию, в Firefox также были введены версии этих методов в виде функций, объявленных непосредственно в конструкторе `Array`. Если использовать эти версии методов, примеры выше можно переписать так:

```
var a = {"0":"a", "1":"b", "2":"c", length:3}; // Объект, подобный массиву
Array.join(a, "+")
Array.slice(a, 0)
Array.map(a, function(x) { return x.toUpperCase(); })
```

Эти статические версии методов массивов чрезвычайно удобны при работе с объектами, подобными массивам, но, так как они не стандартизованы, нельзя рассчитывать, что они будут определены во всех броузерах. В своих программах вы можете использовать следующий программный код, который обеспечит доступность функций перед их использованием:

```
Array.join = Array.join || function(a,sep) {
    return Array.prototype.join.call(a,sep);
};
Array.slice = Array.slice || function(a,from,to) {
    return Array.prototype.slice.call(a,from,to);
};
Array.map = Array.map || function(a, f, thisArg) {
    return Array.prototype.map.call(a, f, thisArg);
}
```

7.12. Строки как массивы

В ECMAScript 5 (и во многих последних версиях броузеров, включая IE8, появившихся до выхода стандарта ECMAScript 5) строки своим поведением напоминают массивы, доступные только для чтения. Вместо метода `charAt()` для обращения к отдельным символам можно использовать квадратные скобки:

```
var s = test;
s.charAt(0)    // => "t"
s[1]           // => "e"
```

Оператор `typeof` для строк все так же возвращает «string», а если строку передать методу `Array.isArray()`, он вернет `false`.

Основное преимущество, которое дает поддержка индексирования строк, – это возможность заменить вызов метода `charAt()` квадратными скобками и получить более краткий, удобочитаемый и, возможно, более эффективный программный код. Однако тот факт, что строки своим поведением напоминают массивы, означает также, что к ним могут применяться универсальные методы массивов. Например:

```
s = "JavaScript"
Array.prototype.join.call(s, " ")      // => "J a v a S c r i p t"
Array.prototype.filter.call(s,         // Фильтровать символы строки
    function(x) {
        return x.match(/[^aeiou]/);    // Совпадение только с согласными
    }).join("")                        // => "JvScrpt"
```

Имейте в виду, что строки являются неизменяемыми значениями, поэтому при работе с ними как с массивами их следует интерпретировать как массивы, доступные только для чтения. Такие методы массивов, как `push()`, `sort()`, `reverse()` и `splice()`, изменяют исходный массив и не будут работать со строками. Однако попытка изменить строку с помощью метода массива не вызовет ошибку: строка просто не изменится.

8

Функции

Функция – это блок программного кода на языке JavaScript, который определяется один раз и может выполняться, или *вызываться*, многократно. Возможно, вы уже знакомы с понятием «функция» под другим названием, таким как *подпрограмма*, или *процедура*. Функции могут иметь *параметры*: определение функции может включать список идентификаторов, которые называются параметрами и играют роль локальных переменных в теле функции. При вызове функций им могут передаваться значения, или *аргументы*, соответствующие их параметрам. Функции часто используют свои аргументы для вычисления *возвращаемого значения*, которое является значением выражения вызова функции. В дополнение к аргументам при вызове любой функции ей передается еще одно значение, определяющее *контекст вызова* – значение в ключевом слове `this`.

Если функция присваивается свойству объекта, она называется *методом* объекта. Когда функция вызывается *посредством объекта*, этот объект становится контекстом вызова, или значением ключевого слова `this`. Функции, предназначенные для инициализации вновь созданных объектов, называются *конструкторами*. Конструкторы были описаны в разделе 6.1, и мы вернемся к ним в главе 9.

Функции в языке JavaScript являются объектами и могут использоваться разными способами. Например, функции могут присваиваться переменным и передаваться другим функциям. Поскольку функции являются объектами, имеется возможность присваивать значения их свойствам и даже вызывать их методы.

В JavaScript допускается создавать определения функций, вложенные в другие функции, и такие функции будут иметь доступ ко всем переменным, присутствующим в области видимости определения. То есть функции в языке JavaScript являются *замыканиями*, что позволяет использовать разнообразные мощные приемы программирования.

8.1. Определение функций

Определение функций выполняется с помощью ключевого слова `function`, которое может использоваться в выражениях определения функций (раздел 4.3) или в инструкциях объявления функций (раздел 5.3.2). В любом случае определение функции начинается с ключевого слова `function`, за которым указываются следующие компоненты:

- Идентификатор, определяющий имя функции. Имя является обязательной частью инструкции объявления функции: оно будет использовано для создания новой переменной, которой будет присвоен объект новой функции. В выражениях определения функций имя может отсутствовать: при его наличии имя будет ссылаться на объект функции только в теле самой функции.
- Пара круглых скобок вокруг списка из нуля или более идентификаторов, разделенных запятыми. Эти идентификаторы будут определять имена параметров функции и в теле функции могут использоваться как локальные переменные.
- Пара фигурных скобок с нулем или более инструкций JavaScript внутри. Эти инструкции составляют тело функции: они выполняются при каждом вызове функции.

В примере 8.1 показано несколько определений функций в виде инструкций и выражений. Обратите внимание, что определения функций в виде выражений удобно использовать, только если они являются частью более крупных выражений, таких как присваивание или вызов функции, которые выполняют некоторые действия с помощью вновь объявленной функции.

Именование функций

В качестве имени функции может использоваться любой допустимый идентификатор. Старайтесь выбирать функциям достаточно описательные, но не длинные имена. Искусство сохранения баланса между краткостью и информативностью приходит с опытом. Правильно подобранные имена функций могут существенно повысить удобочитаемость (а значит, и простоту сопровождения) ваших программ.

Чаще всего в качестве имен функций выбираются глаголы или фразы, начинающиеся с глаголов. По общепринятому соглашению имена функций начинаются со строчной буквы. Если имя состоит из нескольких слов, в соответствии с одним из соглашений они отделяются друг от друга символом подчеркивания, примерно так: `like_this()`, по другому соглашению все слова, кроме первого, начинаются с прописной буквы, примерно так: `likeThis()`. Имена функций, которые, как предполагается, реализуют внутреннюю, скрытую от посторонних глаз функциональность, иногда начинаются с символа подчеркивания.

В некоторых стилях программирования или в четко определенных программных платформах бывает полезно давать наиболее часто используемым функциям очень короткие имена. Примером может служить библиотека jQuery клиентского JavaScript (описываемая в главе 19), в которой широко используется функция с именем `$()` (да-да, просто знак доллара). (В разделе 2.4 уже говорилось, что в идентификаторах JavaScript помимо алфавитно-цифровых символов допускается использовать знаки доллара и подчеркивания.)

Пример 8.1. Определения JavaScript-функций

```
// Выводит имена и значения всех свойств объекта o. Возвращает undefined.
function printprops(o) {
    for(var p in o)
        console.log(p + ": " + o[p] + "\n");
}

// Вычисляет Декартово расстояние между точками (x1,y1) и (x2,y2).
function distance(x1, y1, x2, y2) {
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}

// Рекурсивная функция (вызывающая сама себя), вычисляющая факториал
// Напомню, что x! - это произведение x и всех положительных целых чисел, меньше x.
function factorial(x) {
    if (x <= 1) return 1;
    return x * factorial(x-1);
}

// Следующее выражение определяет функцию, вычисляющую квадрат аргумента.
// Обратите внимание, что она присваивается переменной
var square = function(x) { return x*x; }

// Выражения определения функций могут иметь имена, что позволяет
// производить рекурсивные вызовы.
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };

// Выражения определения функций могут также использоваться в качестве
// аргументов других выражений:
data.sort(function(a,b) { return a-b; });

// Выражения определения функций иногда могут тут же вызываться:
var tensquared = (function(x) {return x*x;}(10));
```

Обратите внимание, что в выражениях определения функций имя функции может отсутствовать. Инструкция объявления функции фактически *объявляет* переменную и присваивает ей объект функции. Выражение определения функции, напротив, не объявляет переменную. Однако в выражениях определения допускается указывать имя функции, как в функции вычисления факториала выше, которое может потребоваться в теле функции для вызова себя самой. Если выражение определения функции включает имя, данное имя будет ссылаться на объект функции в области видимости этой функции. Фактически имя функции становится локальной переменной, доступной только в теле функции. В большинстве случаев имя функции не требуется указывать в выражениях определения, что делает определения более компактными. Особенно удобно использовать выражения для определения однократно используемых функций, как в последних двух примерах выше.

Как описывалось в разделе 5.3.2, инструкции объявления функций «поднимаются» в начало сценария или вмещающей их функции, благодаря чему объявленные таким способом функции могут вызываться в программном коде выше объявления. Это не относится к функциям, которые определяются в виде выражений: чтобы вызвать функцию, необходимо иметь возможность сослаться на нее, однако нельзя сослаться на функцию, которая определяется с помощью выражения, пока она не будет присвоена переменной. Объявления переменных также поднимаются вверх (раздел 3.10.1), но операции присваивания значений этим переменным не поднимаются, поэтому функции, определяемые в виде выражений, не могут вызываться до того, как они будут определены.

Обратите внимание, что большинство (но не все) функций в примере 8.1 содержат инструкцию `return` (раздел 5.6.4). Инструкция `return` завершает выполнение функции и выполняет возврат значения своего выражения (если указано) вызывающей программе. Если выражение в инструкции `return` отсутствует, она возвращает значение `undefined`. Если инструкция `return` отсутствует в функции, интерпретатор просто выполнит все инструкции в теле функции и вернет вызывающей программе значение `undefined`.

Большинство функций в примере 8.1 вычисляют некоторое значение, и в них инструкция `return` используется для возврата этого значения вызывающей программе. Функция `printprops()` несколько отличается в этом смысле: ее работа заключается в том, чтобы вывести имена свойств объекта. Ей не нужно возвращать какое-либо значение, поэтому в функции отсутствует инструкция `return`. Функция `printprops()` всегда будет возвращать значение `undefined`. (Функции, не имеющие возвращаемого значения, иногда называются ***процедурами***.)

8.1.1. Вложенные функции

В JavaScript допускается вложение определений функций в другие функции. Например:

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

Особый интерес во вложенных функциях представляют правила видимости переменных: они могут обращаться к параметрам и переменным, объявленным во вмещающей функции (или функциях). Например, в определении выше внутренняя функция `square()` может читать и изменять параметры `a` и `b`, объявленные во внешней функции `hypotenuse()`. Эти правила видимости, действующие для вложенных функций, играют важную роль, и мы еще вернемся к ним в разделе 8.6.

Как отмечалось в разделе 5.3.2, инструкции объявления функций в действительности не являются настоящими инструкциями, и спецификация ECMAScript допускает использовать их только в программном коде верхнего уровня. Они могут появляться в глобальном программном коде или внутри других функций, но они не могут находиться внутри циклов, условных инструкций, инструкций `try/catch/finally` или `with`.[1] Обратите внимание, что эти ограничения распространяются только на объявления функций в виде инструкции `function`. Выражения определения функций могут присутствовать в любом месте в программе на языке JavaScript.

---

[1] Некоторые реализации JavaScript могут иметь менее строгие требования. Например, в броузере Firefox допускается наличие «условных определений функций» внутри инструкций `if`.

8.2. Вызов функций

Программный код, образующий тело функции, выполняется не в момент определения функции, а в момент ее вызова. Функции в языке JavaScript могут вызываться четырьмя способами:

- как функции,
- как методы,
- как конструкторы и
- косвенно, с помощью их методов `call()` и `apply()`.

8.2.1. Вызов функций

Вызов функций как функций или как методов, выполняется с помощью выражения вызова (раздел 4.5). Выражение вызова состоит из выражения обращения к функции, которое возвращает объект функции, и следующими за ним круглыми скобками со списком из нуля или более выражений-аргументов, разделенных запятыми, внутри. Если выражение обращения к функции является выражением обращения к свойству – если функция является свойством объекта или элементом массива – тогда выражение вызова является выражением вызова метода. Этот случай будет описан ниже. В следующем фрагменте демонстрируется несколько примеров выражений вызова обычных функций:

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

При вызове функции вычисляются все выражения-аргументы (указанные между скобками), и полученные значения используются в качестве аргументов функции. Эти значения присваиваются параметрам, имена которых перечислены в определении функции. В теле функции выражения обращений к параметрам возвращают значения соответствующих аргументов.

При вызове обычной функции возвращаемое функцией значение становится значением выражения вызова. Если возврат из функции происходит по достижении ее конца интерпретатором, возвращается значение `undefined`. Если возврат из функции происходит в результате выполнения инструкции `return`, возвращается значение выражения, следующего за инструкцией `return`, или `undefined`, если инструкция `return` не имеет выражения.

При вызове функции в ECMAScript 3 и в нестрогом режиме ECMAScript 5 контекстом вызова (значением `this`) является глобальный объект. Однако в строгом режиме контекстом вызова является значение `undefined`.

Функции, которые предназначались для использования в виде простых функций, обычно вообще не используют ключевое слово `this`. Впрочем, его можно использовать, чтобы определить, не выполняется ли функция в строгом режиме:

```
// Определение и вызов функции, которая выясняет действующий режим работы.
var strict = (function() { return !this; }());
```

Составление цепочек вызовов методов

Когда методы возвращают объекты, появляется возможность использовать значение, возвращаемое одним методом, как часть последующих вызовов. Это позволяет создавать последовательности («цепочки», или «каскады») вызовов методов в одном выражении. При работе с библиотекой jQuery (глава 19), например, часто можно встретить такие инструкции:

```
// Отыскать все заголовки, отобразить их в значения атрибутов id,
// преобразовать в массив и отсортировать
$(":header").map(function() { return this.id }).get().sort();
```

Если вы пишете метод, не имеющий собственного возвращаемого значения, подумайте о возможности возвращать из него значение `this`. Если неуклонно следовать этому правилу при разработке своего API, появится возможность использовать стиль программирования, известный как составление *цепочек из методов*,[1] когда обращение к имени метода выполняется один раз, а затем может следовать множество вызовов его методов:

```
shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();
```

Не путайте цепочки вызовов методов с цепочками конструкторов, которые описываются в разделе 9.7.2.

8.2.2. Вызов методов

Метод – это не что иное, как функция, которая хранится в виде свойства объекта. Если имеется функция `f` и объект `o`, то можно определить метод объекта `o` с именем `m`, как показано ниже:

```
o.m = f;
```

После этого можно вызвать метод `m()` объекта `o`:

```
o.m();
```

Или, если метод `m()` принимает два аргумента, его можно вызвать так:

```
o.m(x, y);
```

Строка выше является выражением вызова: она включает выражение обращения к функции `o.m` и два выражения-аргумента, `x` и `y`. Выражение обращения к функции в свою очередь является выражением обращения к свойству (раздел 4.4), а это означает, что функция вызывается как метод, а не как обычная функция.

Аргументы и возвращаемое значение при вызове метода обрабатываются точно так же, как при вызове обычной функции. Однако вызов метода имеет одно важное отличие: контекст вызова. Выражение обращения к свойству состоит из двух частей: объекта (в данном случае `o`) и имени свойства (`m`). В подобных выражениях вызова методов объект o становится контекстом вызова, и тело функции получает возможность ссылаться на этот объект с помощью ключевого слова this. Например:

---

[1] Термин был введен Мартином Фаулером (Martin Fowler). См. *http://martinfowler.com/dslwip/MethodChaining.html*.

```
var calculator = { // Литерал объекта
    operand1: 1,
    operand2: 1,
    add: function() {
        // Обратите внимание, что для ссылки на этот объект используется
        // ключевое слово this.
        this.result = this.operand1 + this.operand2;
    }
};
calculator.add();  // Вызвать метод, чтобы вычислить 1+1.
calculator.result  // => 2
```

Чаще всего при вызове методов используется форма обращения к свойствам с помощью оператора точки, однако точно так же можно использовать форму обращения к свойствам с помощью квадратных скобок. Например, оба следующих выражения являются выражениями вызова методов:

```
o["m"](x,y);       // Другой способ записать это выражение: o.m(x,y).
a[0](z)            // Тоже вызов метода (предполагается, что a[0] - это функция).
```

Выражения вызова методов могут включать более сложные выражения обращения к свойствам:

```
customer.surname.toUpperCase(); // Вызвать метод объекта customer.surname
f().m();                        // Вызвать метод m() возвращаемого значения функции f()
```

Методы и ключевое слово this занимают центральное место в парадигме объектно-ориентированного программирования. Любая функция, используемая как метод, фактически получает неявный аргумент – объект, относительно которого она была вызвана. Как правило, методы выполняют некоторые действия с объектом, и синтаксис вызова метода наглядно отражает тот факт, что функция оперирует объектом. Сравните следующие две строки:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

Гипотетически функции, вызывающиеся в этих двух строках, могут производить абсолютно идентичные действия над объектом rect (гипотетическим), но синтаксис вызова метода в первой строке более наглядно демонстрирует, что в центре внимания находится объект rect.

Обратите внимание: this – это именно ключевое слово, а не имя переменной или свойства. Синтаксис JavaScript не допускает возможность присваивания значений элементу this.

В отличие от переменных, ключевое слово this не имеет области видимости, и вложенные функции не наследуют значение this от вызывающей функции. Если вложенная функция вызывается как метод, значением this является объект, относительно которого был сделан вызов. Если вложенная функция вызывается как функция, то значением this будет либо глобальный объект (в нестрогом режиме), либо undefined (в строгом режиме). Распространенная ошибка полагать, что во вложенной функции, которая вызывается как функция, можно использовать `this` для получения доступа к контексту внешней функции. Если во вложенной функции необходимо иметь доступ к значению `this` внешней функции, это значение следует сохранить в переменной, находящейся в области видимости внутренней функции. Для этой цели часто используется переменная с именем `self`. Например:

```
var o = {                          // Объект o.
    m: function() {                // Метод m объекта.
        var self = this;           // Сохранить значение this в переменной.
        console.log(this === o);   // Выведет "true": this - это объект o.
        f();                       // Вызвать вспомогательную ф-цию f().

        function f() {             // Вложенная функция f
            console.log(this === o); // "false": this - глоб. об. или undefined
            console.log(self === o); // "true": self - знач. this внеш. ф-ции.
        }
    }
};
o.m();                             // Вызвать метод m объекта o.
```

В примере 8.5 (раздел 8.7.4) демонстрируется более практичный способ использования идиомы `var self=this`.

8.2.3. Вызов конструкторов

Если вызову функции или метода предшествует ключевое слово `new`, следовательно, это вызов конструктора. (Вызовы конструкторов обсуждались в разделах 4.6 и 6.1.2, а более подробно конструкторы будут рассматриваться в главе 9.) Вызов конструктора отличается от вызова обычной функции или метода особенностями обработки аргументов, контекстом вызова и возвращаемым значением.

Если вызов конструктора включает список аргументов в скобках, эти выражения-аргументы вычисляются и передаются конструктору точно так же, как любой другой функции или методу. Но если конструктор не имеет параметров, синтаксис вызова конструктора в языке JavaScript позволяет вообще опустить скобки и список аргументов. При вызове конструктора всегда можно опустить пару пустых скобок. Например, следующие две строки полностью эквивалентны:

```
var o = new Object();
var o = new Object;
```

Вызов конструктора создает новый пустой объект, наследующий свойство `prototype` конструктора. Назначение функции-конструктора – инициализировать объект, и этот вновь созданный объект передается конструктору как контекст вызова, благодаря чему функция-конструктор может ссылаться на него с помощью ключевого слова `this`. Обратите внимание, что вновь созданный объект передается как контекст вызова, даже если вызов конструктора выглядит как вызов метода. То есть в выражении `new o.m()` контекстом вызова будет вновь созданный объект, а не объект `o`.

Как правило, в функциях-конструкторах не используется инструкция `return`. Обычно они выполняют инициализацию нового объекта и неявно возвращают его, по достижении своего конца. В этом случае значением выражения вызова конструктора становится новый объект. Однако если конструктор явно вернет объект с помощью инструкции `return`, то значением выражения вызова конструктора станет этот объект. Если конструктор выполнит инструкцию `return` без значения или вернет с ее помощью простое значение, это возвращаемое значение будет проигнорировано и в качестве результата вызова будет использован новый объект.

8.2.4. Косвенный вызов

Функции в языке JavaScript являются объектами и подобно другим объектам имеют свои методы. В их числе есть два метода, `call()` и `apply()`, выполняющие косвенный вызов функции. Оба метода позволяют явно определить значение `this` для вызываемой функции, что дает возможность вызывать любую функцию как метод любого объекта, даже если фактически она не является методом этого объекта. Кроме того, обоим методам можно передать аргументы вызова. Метод `call()` позволяет передавать аргументы для вызываемой функции в своем собственном списке аргументов, а метод `apply()` принимает массив значений, которые будут использованы как аргументы. Подробнее о методах `call()` и `apply()` рассказывается в разделе 8.7.3.

8.3. Аргументы и параметры функций

В языке JavaScript, в определениях функций не указываются типы параметров, а при вызове функций не выполняется никаких проверок типов передаваемых значений аргументов. Фактически при вызове функций в языке JavaScript не проверяется даже количество аргументов. В подразделах ниже описывается, что происходит, если число аргументов в вызове функции меньше или больше числа объявленных параметров. В них также демонстрируется, как можно явно проверить типы аргументов функции, если необходимо гарантировать, что функция не будет вызвана с некорректными аргументами.

8.3.1. Необязательные аргументы

Когда число аргументов в вызове функции меньше числа объявленных параметров, недостающие аргументы получают значение `undefined`. Часто бывает удобным писать функции так, чтобы некоторые аргументы были необязательными и могли опускаться при вызове функции. В этом случае желательно предусмотреть возможность присваивания достаточно разумных значений по умолчанию параметрам, которые могут быть опущены. Например:

```
// Добавить в массив a перечислимые имена свойств объекта o и вернуть его.
// Если аргумент a не не был передан, создать и вернуть новый массив.
function getPropertyNames(o, /* необязательный */ a) {
    if (a === undefined) a = []; // Если массив не определен, создать новый
    for(var property in o) a.push(property);
    return a;
}

// Эта функция может вызываться с 1 или 2 аргументами:
var a = getPropertyNames(o); // Получить свойства объекта o в новом массиве
getPropertyNames(p,a);       // добавить свойства объекта p в этот массив
```

Вместо инструкции `if` в первой строке этой функции можно использовать оператор `||` следующим образом:

```
a = a || [];
```

В разделе 4.10.2 говорилось, что оператор `||` возвращает первый аргумент, если он имеет истинное значение, и в противном случае возвращает второй аргумент. В данном примере, если во втором аргументе будет передан какой-либо объект, функция будет использовать его. Но если второй аргумент отсутствует (или в нем будет передано значение `null`), будет использоваться вновь созданный массив.

Обратите внимание, что при объявлении функций необязательные аргументы должны завершать список аргументов, чтобы их можно было опустить. Программист, который будет писать обращение к вашей функции, не сможет передать второй аргумент и при этом опустить первый: он будет вынужден явно передать в первом аргументе значение `undefined`. Обратите также внимание на комментарий `/* необязательный */` в определении функции, который подчеркивает тот факт, что параметр является необязательным.

8.3.2. Списки аргументов переменной длины: объект Arguments

Если число аргументов в вызове функции превышает число имен параметров, функция лишается возможности напрямую обращаться к неименованным значениям. Решение этой проблемы предоставляет объект `Arguments`. В теле функции идентификатор `arguments` ссылается на объект `Arguments`, присутствующий в вызове. Объект `Arguments` – это объект, подобный массиву (раздел 7.11), позволяющий извлекать переданные функции значения по их номерам, а не по именам.

Предположим, что была определена функция `f`, которая требует один аргумент, `x`. Если вызвать эту функцию с двумя аргументами, то первый будет доступен внутри функции по имени параметра `x` или как `arguments[0]`. Второй аргумент будет доступен только как `arguments[1]`. Кроме того, подобно настоящим массивам, `arguments` имеет свойство `length`, определяющее количество содержащихся элементов. То есть в теле функции `f`, вызываемой с двумя аргументами, `arguments.length` имеет значение 2.

Объект `Arguments` может использоваться с самыми разными целями. Следующий пример показывает, как с его помощью проверить, была ли функция вызвана с правильным числом аргументов, – ведь JavaScript этого за вас не сделает:

```
function f(x, y, z)
{
    // Сначала проверяется, правильное ли количество аргументов передано
    if (arguments.length != 3) {
        throw new Error("функция f вызвана с " + arguments.length +
                        "аргументами, а требуется 3.");
    }
    // А теперь сам код функции...
}
```

Обратите внимание, что зачастую нет необходимости проверять количество аргументов, как в данном примере. Поведение по умолчанию интерпретатора Java-

Script отлично подходит для большинства случаев: отсутствующие аргументы замещаются значением `undefined`, а лишние аргументы просто игнорируются.

Объект `Arguments` иллюстрирует важную возможность JavaScript-функций: они могут быть написаны таким образом, чтобы работать с любым количеством аргументов. Следующая функция принимает любое число аргументов и возвращает значение самого большого из них (аналогично ведет себя встроенная функция `Math.max()`):

```
function max(/*...*/)
{
    var m = Number.NEGATIVE_INFINITY;
    // Цикл по всем аргументам, поиск и сохранение наибольшего из них
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > max) max = arguments[i];
    // Вернуть наибольшее значение
    return max;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6); // => 10000
```

Функции, подобные этой и способные принимать произвольное число аргументов, называются *функциями с переменным числом аргументов* (*variadic functions*, *variable arity functions* или *varargs functions*). Этот термин возник вместе с появлением языка программирования С.

Обратите внимание, что функции с переменным числом аргументов не должны допускать возможность вызова с пустым списком аргументов. Будет вполне разумным использовать объект `arguments[]` при написании функции, ожидающей получить фиксированное число обязательных именованных аргументов, за которыми может следовать произвольное число необязательных неименованных аргументов.

Не следует забывать, что `arguments` фактически не является массивом – это объект `Arguments`. В каждом объекте `Arguments` имеются пронумерованные элементы массива и свойство `length`, но с технической точки зрения это не массив. Лучше рассматривать его как объект, имеющий некоторые пронумерованные свойства. Подробнее об объектах, подобных массивам, рассказывается в разделе 7.11.

У объекта `Arguments` есть одна *очень* необычная особенность. Когда у функции имеются именованные параметры, элементы массива объекта `Arguments` при выполнении в нестрогом режиме являются синонимами параметров, содержащих аргументы функции. Массив `arguments[]` и имена параметров – это два разных средства обращения к одним и тем же переменным. Изменение значения аргумента через имя аргумента меняет значение, извлекаемое через массив `arguments[]`. Изменение значения аргумента через массив `arguments[]` меняет значение, извлекаемое по имени аргумента. Например:

```
function f(x) {
    console.log(x);         // Выведет начальное значение аргумента
    arguments[0] = null;    // При изменении элемента массива изменяется x!
    console.log(x);         // Теперь выведет "null"
}
```

Определенно, это не совсем то поведение, которое можно было бы ожидать от настоящего массива. В этом случае `arguments[0]` и `x` могли бы ссылаться на одно и то же значение, но изменение одной ссылки не должно оказывать влияния на другую.

Эта особенность в поведении объекта `Arguments` была ликвидирована в строгом режиме, предусматриваемом стандартом ECMAScript 5. Кроме того, в строгом режиме имеется еще несколько отличий. В нестрогом режиме `arguments` – это всего лишь обычный JavaScript-идентификатор, а не зарезервированное слово. В строгом режиме не допускается использовать имя `arguments` в качестве имени параметра или локальной переменной функции и отсутствует возможность присваивать значения элементам `arguments`.

8.3.2.1. Свойства callee и caller

Помимо элементов своего массива объект `Arguments` определяет свойства `callee` и `caller`. При попытке изменить значения этих свойств в строгом режиме ECMAScript 5 гарантированно возбуждается исключение `TypeError`. Однако в нестрогом режиме стандарт ECMAScript утверждает, что свойство `callee` ссылается на выполняемую в данный момент функцию. Свойство `caller` не является стандартным, но оно присутствует во многих реализациях и ссылается на функцию, вызвавшую текущую. Свойство `caller` можно использовать для доступа к стеку вызовов, а свойство `callee` особенно удобно использовать для рекурсивного вызова неименованных функций:

```
var factorial = function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x-1);
};
```

8.3.3. Использование свойств объекта в качестве аргументов

Когда функция имеет более трех параметров, становится трудно запоминать правильный порядок их следования. Чтобы предотвратить ошибки и избавить программиста от необходимости заглядывать в документацию всякий раз, когда он намеревается вставить в программу вызов такой функции, можно предусмотреть возможность передачи аргументов в виде пар имя/значение в произвольном порядке. Чтобы реализовать такую возможность, при определении функции следует предусмотреть передачу объекта в качестве единственного аргумента. Благодаря такому стилю пользователи функции смогут передавать функции объект, в котором будут определяться необходимые пары имя/значение. В следующем фрагменте приводится пример такой функции, а также демонстрируется возможность определения значений по умолчанию для опущенных аргументов:

```
// Скопировать length элементов из массива from в массив to.
// Копирование начинается с элемента from_start в массиве from
// и выполняется в элементы, начиная с to_start в массиве to.
// Запомнить порядок следования аргументов такой функции довольно сложно.
function arraycopy(/* массив */ from, /* индекс */ from_start,
                   /* массив */ to,   /* индекс */ to_start,
                   /* целое */ length)
{
    // здесь находится реализация функции
}

// Эта версия функции чуть менее эффективная, но не требует запоминать порядок следования
// аргументов, а аргументы from_start и to_start по умолчанию принимают значение 0.
function easycopy(args) {
    arraycopy(args.from,
              args.from_start || 0, // Обратите внимание, как назначаются
              args.to,              // значения по умолчанию
              args.to_start || 0,
              args.length);
}
// Далее следует пример вызова функции easycopy():
var a = [1,2,3,4], b = [];
easycopy({from: a, to: b, length: 4});
```

8.3.4. Типы аргументов

В языке JavaScript параметры функций объявляются без указания их типов, а во время передачи значений функциям не производится никакой проверки их типов. Вы можете сделать свой программный код самодокументируемым, выбирая описательные имена для параметров функций и включая описание типов аргументов в комментарии, как это сделано в только что рассмотренном примере функции `arraycopy()`. Для необязательных аргументов в комментарий можно добавлять слово «необязательный» («optional»). А если функция может принимать произвольное число аргументов, можно использовать многоточие:

```
function max(/* число... */) { /* тело функции */ }
```

Как отмечалось в разделе 3.8, при необходимости JavaScript выполняет преобразование типов. Таким образом, если определить функцию, которая ожидает получить строковый аргумент, а затем вызвать ее с аргументом какого-нибудь другого типа, значение аргумента просто будет преобразовано в строку, когда функция пытается обратиться к нему как к строке. В строку может быть преобразовано любое простое значение, и все объекты имеют методы `toString()` (правда, не всегда полезные); тем самым устраняется вероятность появления ошибки.

Однако такой подход может использоваться не всегда. Вернемся к методу `arraycopy()`, продемонстрированному выше. Он ожидает получить массив в первом аргументе. Любое обращение к функции окажется неудачным, если первым аргументом будет не массив (или, возможно, объект, подобный массиву). Если функция должна вызываться чаще, чем один-два раза, следует добавить в нее проверку соответствия типов аргументов. Гораздо лучше сразу же прервать вызов функции в случае передачи аргументов ошибочных типов, чем продолжать выполнение, которое потерпит неудачу с сообщением об ошибке, запутывающим ситуацию. Ниже приводится пример функции, выполняющей проверку типов. Обратите внимание, что она использует функцию `isArrayLike()` из раздела 7.11:

```
// Возвращает сумму элементов массива (или объекта, подобного массиву) a.
// Все элементы массива должны быть числовыми, при этом значения null
// и undefined игнорируются.
function sum(a) {
    if (isArrayLike(a)) {
        var total = 0;
        for(var i = 0; i < a.length; i++) { // Цикл по всем элементам
            var element = a[i];
            if (element == null) continue;       // Пропустить null и undefined
            if (isFinite(element)) total += element;
            else throw new Error("sum(): все элементы должны быть числами");
        }
        return total;
    }
    else throw new Error("sum(): аргумент должен быть массивом");
}
```

Этот метод sum() весьма строго относится к проверке типов входных аргументов и генерирует исключения с достаточно информативными сообщениями, если типы входных аргументов не соответствуют ожидаемым. Тем не менее он остается достаточно гибким, обслуживая наряду с настоящими массивами объекты, подобные массивам, и игнорируя элементы, имеющие значения null и undefined.

JavaScript – чрезвычайно гибкий и к тому же слабо типизированный язык, благодаря чему можно писать функции, которые достаточно терпимо относятся к количеству и типам входных аргументов. Далее приводится метод flexsum(), реализующий такой подход (и, вероятно, являющийся примером другой крайности). Например, он принимает любое число входных аргументов и рекурсивно обрабатывает те из них, которые являются массивами. Вследствие этого он может принимать переменное число аргументов или массив аргументов. Кроме того, он прилагает максимум усилий, чтобы преобразовать нечисловые аргументы в числа, прежде чем сгенерировать исключение:

```
function flexisum(a) {
    var total = 0;
    for(var i = 0; i < arguments.length; i++) {
        var element = arguments[i], n;
        if (element == null) continue; // Игнорировать null и undefined
        if (isArray(element))          // Если аргумент - массив
            n = flexisum.apply(this, element); // вычислить сумму рекурсивно
        else if (typeof element === "function") // Иначе, если это функция...
            n = Number(element());              // вызвать и преобразовать.
        else n = Number(element);       // Иначе попробовать преобразовать

        if (isNaN(n)) // Если не удалось преобразовать в число, возбудить искл.
            throw Error("flexisum(): невозможно преобразовать " + element +
                        " в число");
        total += n;   // Иначе прибавить n к total
    }
    return total;
}
```

8.4. Функции как данные

Самые важные особенности функций заключаются в том, что они могут определяться и вызываться. Определение и вызов функции – это синтаксические средства JavaScript и большинства других языков программирования. Однако в JavaScript функции – это не только синтаксические конструкции, но и данные, а это означает, что они могут присваиваться переменным, храниться в свойствах объектов или элементах массивов, передаваться как аргументы функциями и т. д.[1]

Чтобы понять, как функции в JavaScript могут быть одновременно синтаксическими конструкциями и данными, рассмотрим следующее определение функции:

```
function square(x) { return x*x; }
```

Это определение создает новый объект функции и присваивает его переменной `square`. Имя функции действительно нематериально – это просто имя переменной, которая ссылается на объект функции. Функция может быть присвоена другой переменной, и при этом работать так же, как и раньше:

```
var s = square; // Теперь s ссылается на ту же функцию, что и square
square(4);      // => 16
s(4);           // => 16
```

Функции могут быть также присвоены не только глобальным переменным, но и свойствам объектов. В этом случае их называют методами:

```
var o = {square: function(x) { return x*x; }}; // Литерал объекта
var y = o.square(16);                          // y = 256
```

Функции могут быть даже безымянными, например, в случае присваивания их элементам массива:

```
var a = [function(x) { return x*x; }, 20];    // Литерал объекта
a[0](a[1]);                                   // => 400
```

Синтаксис вызова функции в последнем примере выглядит необычно, однако это вполне допустимый вариант применения выражения вызова!

В примере 8.2 демонстрируется, что можно делать, когда функции выступают в качестве данных. Хотя пример может показаться вам несколько сложным, комментарии объясняют, что происходит.

Пример 8.2. Использование функций как данных

```
// Определения нескольких простых функций
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Эта функция принимает одну из предыдущих функций
// в качестве аргумента и вызывает ее с двумя операндами
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// Так можно вызвать эту функцию для вычисления выражения (2+3)+(4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));
```

---

[1] Это может показаться не столь интересным, если вы не знакомы с такими языками, как Java, в которых функции являются частью программы, но не могут управляться программой.

```
// Ради примера реализуем эти функции снова, на этот раз
// с помощью литералов функций внутри литерала объекта.
var operators = {
    add:      function(x,y) { return x+y; },
    subtract: function(x,y) { return x-y; },
    multiply: function(x,y) { return x*y; },
    divide:   function(x,y) { return x/y; },
    pow:      Math.pow  // Можно использовать даже предопределенные функции
};

// Эта функция принимает имя оператора, отыскивает оператор в объекте,
// а затем вызывает его с указанными операндами.
// Обратите внимание на синтаксис вызова функции оператора.
function operate2(operation, operand1, operand2)
{
    if (typeof operators[operation] === "function")
        return operators[operation](operand1, operand2);
    else throw "неизвестный оператор";
}

// Вычислить значение ("hello" + " " + "world"):
var j = operate2("add", "hello", operate2("add", " ", "world"));

// Использовать предопределенную функцию Math.pow():
var k = operate2("pow", 10, 2);
```

В качестве еще одного примера использования функций как значений рассмотрим метод `Array.sort()`. Он сортирует элементы массива. Существует много возможных порядков сортировки (числовой, алфавитный, по датам, по возрастанию, по убыванию и т. д.), поэтому метод `sort()` принимает в качестве необязательного аргумента функцию, которая сообщает о том, как выполнять сортировку. Эта функция делает простую работу: получает два значения, сравнивает их и возвращает результат, указывающий, какой из элементов должен быть первым. Этот аргумент-функция делает метод `Array.sort()` совершенно универсальным и бесконечно гибким – он может сортировать любой тип данных в любом мыслимом порядке. Примеры использования функции `Array.sort()` представлены в разделе 7.8.3.

8.4.1. Определение собственных свойств функций

Функции в языке JavaScript являются не простыми значениями, а особой разновидностью объектов, благодаря чему функции могут иметь свойства. Когда функции требуется «статическая» переменная, значение которой должно сохраняться между ее вызовами, часто оказывается удобным использовать свойство объекта функции, позволяющее не занимать пространство имен определениями глобальных переменных. Например, предположим, что надо написать функцию, возвращающую уникальное целое число при каждом своем вызове. Функция никогда не должна возвращать одно и то же значение дважды. Чтобы обеспечить это, функция должна запоминать последнее возвращенное значение и сохранять его между ее вызовами. Эту информацию можно было бы хранить в глобальной переменной, но это было бы нежелательно, потому что данная информация используется только этой функцией. Лучше сохранять ее в свойстве объекта `Function`. Вот пример функции, которая возвращает уникальное целое значение при каждом вызове:

```
// Инициализировать свойство counter объекта функции. Объявления функций
// поднимаются вверх, поэтому мы можем выполнить это присваивание до объявления функции.
uniqueInteger.counter = 0;

// Эта функция возвращает разные целые числа при каждом вызове.
// Для сохранения следующего возвращаемого значения она использует собственное свойство.
function uniqueInteger() {
    return uniqueInteger.counter++; // Увеличить и вернуть свойство counter
}
```

Еще один пример, взгляните на следующую функцию `factorial()`, которая использует собственные свойства (интерпретируя себя как массив) для сохранения результатов предыдущих вычислений:

```
// Вычисляет факториалы и сохраняет результаты в собственных свойствах.
function factorial(n) {
    if (isFinite(n) && n>0 && n==Math.round(n)) { // Только конечные положительные целые
        if (!(n in factorial))                   // Если не сохранялось ранее
            factorial[n] = n * factorial(n-1);   // Вычислить и сохранить
        return factorial[n];                     // Вернуть сохр. результат
    }
    else return NaN;                             // Для ошибочного аргумента
}
factorial[1] = 1; // Инициализировать кэш базовым случаем.
```

8.5. Функции как пространства имен

В разделе 3.10.1 говорилось, что в языке JavaScript существует такое понятие, как область видимости функции: переменные, объявленные внутри функции, видимы в любой точке функции (включая и вложенные функции), но они существуют только в пределах функции. Переменные, объявленные за пределами функции, являются глобальными переменными и видимы в любой точке JavaScript-программы. В языке JavaScript отсутствует возможность объявлять переменные, доступные только внутри отдельно расположенного блока программного кода, и по этой причине иногда бывает удобно определять функции, которые играют роль временного пространства имен, где можно объявлять переменные, не засоряя глобальное пространство имен.

Предположим, например, что имеется модуль JavaScript, который можно использовать в различных JavaScript-программах (или, если говорить на языке клиентского JavaScript, в различных веб-страницах). Допустим, что в программном коде этого модуля, как в практически любом другом программном коде, объявляются переменные для хранения промежуточных результатов вычислений. Проблема состоит в том, что модуль будет использоваться множеством различных программ, и поэтому заранее неизвестно, будут ли возникать конфликты между переменными, создаваемыми модулем, и переменными, используемыми в программах, импортирующих этот модуль. Решение состоит в том, чтобы поместить программный код модуля в функцию и затем вызывать эту функцию. При таком подходе переменные модуля из глобальных превратятся в локальные переменные функции:

```
function mymodule() {
    // Здесь находится реализация модуля.
    // Любые переменные, используемые модулем, превратятся в локальные
    // переменные этой функции и не будут засорять глобальное пространство имен.
}
mymodule(); // Но не забудьте вызвать функцию!
```

Данный программный код объявляет единственную глобальную переменную: имя функции «mymodule». Если даже единственное имя это слишком много, можно определить и вызвать анонимную функцию в одном выражении:

```
(function() { // функция mymodule переписана как неименованное выражение
    // Здесь находится реализация модуля.
}());         // конец литерала функции и ее вызов.
```

Такой способ определения и вызова функции в одном выражении настолько часто используется в практике, что превратился в типичный прием. Обратите внимание на использование круглых скобок в этом примере. Открывающая скобка перед ключевым словом `function` **является обязательной, потому что без нее ключевое слово** `function` **будет интерпретироваться как инструкция объявления функции. Скобки позволят интерпретатору распознать этот фрагмент как выражение определения функции. Это обычный способ – заключение в скобки, даже когда они не требуются, определения функции, которая должна быть вызвана сразу же после ее определения.**

Практическое применение приема создания пространства имен демонстрируется в примере 8.3. Здесь определяется анонимная функция, возвращающая функцию `extend()`**, подобную той, что была представлена в примере 6.2. Анонимная функция проверяет наличие хорошо известной ошибки в Internet Explorer и возвращает исправленную версию функции, если это необходимо. Помимо этого, анонимная функция играет роль пространства имен, скрывающего массив с именами свойств.**

Пример 8.3. Функция extend(), исправленная, если это необходимо

```
// Определяет функцию extend, которая копирует свойства второго и последующих аргументов
// в первый аргумент. Здесь реализован обход ошибки в IE: во многих версиях IE цикл for/in
// не перечисляет перечислимые свойства объекта o, если одноименное свойство
// его прототипа является неперечислимым. Это означает, что такие свойства,
// как toString, обрабатываются некорректно, если явно не проверять их.
var extend = (function() { // Присвоить значение, возвращаемое этой функцией
    // Сначала проверить наличие ошибки, прежде чем исправлять ее.
    for(var p in {toString:null}) {
        // Если мы оказались здесь, значит, цикл for/in работает корректно
        // и можно вернуть простую версию функции extend()
        return function extend(o) {
            for(var i = 1; i < arguments.length; i++) {
                var source = arguments[i];
                for(var prop in source) o[prop] = source[prop];
            }
            return o;
        };
    }
    // Если мы оказались здесь, следовательно, цикл for/in не перечислил
    // свойство toString тестового объекта. Поэтому необходимо вернуть версию extend(),
    // которая явно проверяет неперечислимость свойств прототипа Object.prototype.

    // Список свойств, которые необходимо проверить
    var protoprops = ["toString", "valueOf", "constructor", "hasOwnProperty",
                      "isPrototypeOf", "propertyIsEnumerable", "toLocaleString"];

    return function patched_extend(o) {
        for(var i = 1; i < arguments.length; i++) {
            var source = arguments[i];
            // Скопировать все перечислимые свойства
            for(var prop in source) o[prop] = source[prop];
            // А теперь проверить специальные случаи свойств
            for(var j = 0; j < protoprops.length; j++) {
                prop = protoprops[j];
                if (source.hasOwnProperty(prop)) o[prop] = source[prop];
            }
        }
        return o;
    };
}());
```

8.6. Замыкания

Как и в большинстве языков программирования, в JavaScript используются *лексические области видимости*. Это означает, что при выполнении функций действуют области видимости переменных, которые имелись на момент их определения, а не на момент вызова. Для реализации лексической области видимости внутренняя информация о состоянии объекта функции в языке JavaScript должна включать не только программный код функции, но еще и ссылку на текущую цепочку областей видимости. (Прежде чем продолжить чтение этого раздела, вам, возможно, следует повторно прочитать сведения об областях видимости переменных и цепочках областей видимости в разделах 3.10 и 3.10.3.) Такая комбинация объекта функции и области видимости (множества связанных переменных), в которой находятся переменные, используемые вызываемой функцией, в литературе по информационным технологиям называется *замыканием*.[1]

Технически все функции в языке JavaScript образуют замыкания: они являются объектами и имеют ассоциированные с ними цепочки областей видимости. Большинство функций вызываются внутри той же цепочки областей видимости, которая действовала на момент определения функции, и в этой ситуации факт образования замыкания не имеет никакого значения. Интересные особенности замыканий начинают проявляться, когда их вызов производится в другой цепочке областей видимости, отличной от той, что действовала на момент определения. Чаще всего это происходит, когда объект вложенной функции возвращается функцией, вмещающей ее определение. Существует множество мощных приемов программирования, вовлекающих такого рода вложенные функции-замыкания, и их использование довольно широко распространено в программировании на языке JavaScript. Замыкания могут выглядеть малопонятными при первом знакомстве, однако вам необходимо хорошо понимать их, чтобы чувствовать себя уверенно при их использовании.

---

[1] Это старый термин, который отражает тот факт, что переменные функции привязаны к цепочке областей видимости, вследствие чего функция образует «замыкание» по своим переменным.

Первый шаг к пониманию замыканий – знакомство с правилами лексической области видимости, действующими для вложенных функций. Взгляните на следующий пример (который напоминает пример в разделе 3.10):

```
var scope = "global scope";        // Глобальная переменная
function checkscope() {
    var scope = "local scope";     // Локальная переменная
    function f() { return scope; } // Вернет значение локальной переменной scope
    return f();
}
checkscope()                       // => "local scope"
```

Реализация замыканий

Понять суть замыканий будет совсем несложно, если усвоить правило лексической области видимости: во время выполнения функции используется цепочка областей видимости, которая действовала в момент ее определения. Однако некоторые программисты испытывают сложности при освоении замыканий, потому что не до конца понимают особенности реализации. Известно, думают они, что локальные переменные, объявленные во внешней функции, прекращают свое существование после выхода из внешней функции, но тогда как вложенная функция может использовать цепочку областей видимости, которая больше не существует? Если вы задавали себе такой вопрос, значит, у вас наверняка есть опыт работы с низкоуровневыми языками программирования, такими как C, и аппаратными архитектурами, использующими стек: если локальные переменные размещать на стеке, они действительно прекращают свое существование после завершения функции.

Но вспомните определение цепочки областей видимости из раздела 3.10.3. Там она описывалась как список объектов, а не стек. Каждый раз, когда интерпретатор JavaScript вызывает функцию, он создает новый объект для хранения локальных переменных этой функции, и этот объект добавляется в цепочку областей видимости. Когда функция возвращает управление, этот объект удаляется из цепочки. Если в программе нет вложенных функций и нет ссылок на этот объект, он будет утилизирован сборщиком мусора. Если в программе имеются вложенные функции, тогда каждая из этих функций будет владеть ссылкой на свою цепочку областей видимости, а цепочка будет ссылаться на объекты с локальными переменными. Если объекты вложенных функций существуют только в пределах своих внешних функций, они сами будут утилизированы сборщиком мусора, а вместе с ними будут утилизированы и объекты с локальными переменными, на которые они ссылались. Но если функция определяет вложенную функцию и возвращает ее или сохраняет в свойстве какого-либо объекта, то образуется внешняя ссылка на вложенную функцию. Такой объект вложенной функции не будет утилизирован сборщиком мусора, и точно так же не будет утилизирован объект с локальными переменными, на который она ссылается.

Функция checkscope() объявляет локальную переменную и вызывает функцию, возвращающую значение этой переменной. Должно быть совершенно понятно, почему вызов checkscope() возвращает строку «local scope». Теперь немного изменим пример. Сможете ли вы сказать, какое значение вернет этот фрагмент?

```
var scope = "global scope";          // Глобальная переменная
function checkscope() {
    var scope = "local scope";       // Локальная переменная
    function f() { return scope; }   // Вернет значение локальной переменной scope
    return f;
}
checkscope()()                       // Какое значение вернет этот вызов?
```

В этой версии пара круглых скобок была перемещена из тела функции checkscope() за ее пределы. Вместо вызова вложенной функции и возврата ее результата checkscope() теперь просто возвращает сам объект вложенной функции. Что произойдет, если вызвать вложенную функцию (добавив вторую пару скобок в последней строке примера) из-за пределов функции, в которой она определена?

Напомню главное правило лексической области видимости: при выполнении функции в языке JavaScript используется цепочка областей видимости, действовавшая на момент ее определения. Вложенная функция f() была определена в цепочке видимости, где переменная scope связана со значением «local scope». Эта связь остается действовать и при выполнении функции f, независимо от того, откуда был произведен ее вызов. Поэтому последняя строка в примере выше вернет «local scope», а не «global scope». Проще говоря, эта особенность является самой удивительной и мощной чертой замыканий: они сохраняют связь с локальными переменными (и параметрами) внешней функции, где они были определены.

В разделе 8.4.1 был приведен пример функции uniqueInteger(), в которой используется свойство самой функции для сохранения следующего возвращаемого значения. Недостаток такого решения состоит в том, что ошибочный или злонамеренный программный код может сбросить значение счетчика или записать в него нечисловое значение, вынудив функцию uniqueInteger() нарушить обязательство возвращать «уникальное» или «целочисленное» значение. Замыкания запирают локальные переменные в объекте вызова функции и могут использовать эти переменные для хранения частной информации. Ниже показано, как можно реализовать функцию uniqueInteger() с использованием замыкания:

```
var uniqueInteger = (function() {          // Определение и вызов
                        var counter = 0; // Частное значение для функции ниже
                        return function() { return counter++; };
                    }());
```

Внимательно изучите этот пример, чтобы понять, как он действует. На первый взгляд, первая строка выглядит как инструкция присваивания функции переменной uniqueInteger. Фактически же это определение и вызов функции (как подсказывает открывающая круглая скобка в первой строке), поэтому в действительности переменной uniqueInteger присваивается значение, возвращаемое функцией. Если теперь обратить внимание на тело функции, можно увидеть, что она возвращает другую функцию. Именно этот объект вложенной функции и присваивается переменной uniqueInteger. Вложенная функция имеет доступ к переменным в ее области видимости и может использовать переменную counter, объявленную во внешней функции. После возврата из внешней функции никакой другой программный код не будет иметь доступа к переменной `counter`: вложенная функция будет обладать исключительным правом доступа к ней.

Скрытые переменные, такие как `counter`, не являются исключительной собственностью единственного замыкания: в одной и той же внешней функции вполне возможно определить две или более вложенных функций, которые будут совместно использовать одну цепочку областей видимости. Рассмотрим следующий пример:

```
function counter() {
    var n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}

var c = counter(), d = counter(); // Создать два счетчика
c.count()                         // => 0
d.count()                         // => 0: они действуют независимо
c.reset()                         // методы reset() и count() совместно
                                  // используют одну переменную
c.count()                         // => 0: сброс счетчика c
d.count()                         // => 1: не оказывает влияния на счетчик d
```

Функция `counter()` возвращает объект «счетчика». Этот объект имеет два метода: `count()`, возвращающий следующее целое число, и `reset()`, сбрасывающий счетчик в начальное состояние. В первую очередь следует запомнить, что два метода совместно используют одну и ту же частную переменную `n`. Во-вторых, каждый вызов функции `counter()` создает новую цепочку областей видимости и новую скрытую переменную. То есть, если вызвать функцию `counter()` дважды, она вернет два объекта-счетчика с различными скрытыми переменными. Вызов методов `count()` и `reset()` одного объекта-счетчика не оказывает влияния на другой.

Важно отметить, что описанный прием образования замыканий можно использовать в комбинации с приемом определения свойств с методами доступа. Следующая версия функции `counter()` является вариацией примера, представленного в разделе 6.6, но здесь для хранения скрытой информации вместо обычного свойства объекта используются замыкания:

```
function counter(n) { // Аргумент n функции - скрытая переменная
    return {
        // Метод чтения свойства возвращает и увеличивает переменную счетчика.
        get count() { return n++; },
        // Метод записи в свойство не позволяет уменьшать значение n
        set count(m) {
            if (m >= n) n = m;
            else throw Error("значение счетчика нельзя уменьшить ");
        }
    };
}

var c = counter(1000);
c.count            // => 1000
```

```
c.count          // => 1001
c.count = 2000
c.count          // => 2000
c.count = 2000   // => Ошибка!
```

Обратите внимание, что эта версия функции `counter()` не объявляет локальную переменную. Для сохранения информации она просто использует параметр `n`, доступный обоим методам доступа к свойству. Это позволяет программе, вызывающей `counter()`, определять начальное значение скрытой переменной.

В примере 8.4 демонстрируется обобщение приема совместного использования скрытой информации в замыканиях. Этот пример определяет функцию `addPrivateProperty()`, которая в свою очередь определяет скрытую переменную и две вложенные функции для чтения и записи значения этой переменной. Она добавляет эти вложенные функции как методы указанного вами объекта:

Пример 8.4. Реализация методов доступа к частному свойству с использованием замыканий

```
// Эта функция добавляет методы доступа к свойству с заданным именем объекта o.
// Методы получают имена вида get<name> и set<name>. Если дополнительно предоставляется
// функция проверки, метод записи будет использовать ее для проверки значения
// перед сохранением. Если функция проверки возвращает false,
// метод записи генерирует исключение.
//
// Необычность такого подхода заключается в том, что значение свойства,
// доступного методам, сохраняется не в виде свойства объекта o, а в виде
// локальной переменной этой функции. Кроме того, методы доступа также определяются
// внутри этой функции и потому получают доступ к этой локальной переменной.
// Это означает, что значение доступно только этим двум методам и не может быть
// установлено или изменено иначе, как методом записи.
function addPrivateProperty(o, name, predicate) {
    var value; // Это значение свойства

    // Метод чтения просто возвращает значение.
    o["get" + name] = function() { return value; };

    // Метод записи сохраняет значение или возбуждает исключение,
    // если функция проверки отвергает это значение.
    o["set" + name] = function(v) {
        if (predicate && !predicate(v))
            throw Error("set" + name + ": недопустимое значение " + v);
        else
            value = v;
    };
}

// Следующий фрагмент демонстрирует работу метода addPrivateProperty().
var o = {};    // Пустой объект

// Добавить к свойству методы доступа с именами getName() и setName()
// Обеспечить допустимость только строковых значений
addPrivateProperty(o, "Name", function(x) { return typeof x == "string"; });

o.setName("Frank");        // Установить значение свойства
console.log(o.getName()); // Получить значение свойства
o.setName(0);              // Попробовать установить значение свойства неверного типа
```

Мы увидели несколько примеров, когда замыкания определяются в одной и той же цепочке областей видимости и совместно используют одну локальную переменную или переменные. Важно знать и уметь пользоваться этим приемом, но не менее важно уметь распознавать ситуации, когда замыкания получают переменную в совместное использование по ошибке. Рассмотрим следующий пример:

```
// Эта функция возвращает функцию, которая всегда возвращает v
function constfunc(v) { return function() { return v; }; }

// Создать массив функций-констант:
var funcs = [];
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// Функция в элементе массива с индексом 5 возвращает 5.
funcs[5]() // => 5
```

При создании подобного программного кода, который создает множество замыканий в цикле, часто допускают ошибку, помещая цикл внутрь функции, которая определяет замыкания. Например, взгляните на следующий фрагмент:

```
// Возвращает массив функций, возвращающих значения 0-9
function constfuncs() {
    var funcs = [];
    for(var i = 0; i < 10; i++)
        funcs[i] = function() { return i; };
    return funcs;
}

var funcs = constfuncs();
funcs[5]()    // Что вернет этот вызов?
```

Функция выше создает 10 замыканий и сохраняет их в массиве. Замыкания образуются в одном и том же вызове функции, поэтому все они получат доступ к переменной `i`. Когда `constfuncs()` вернет управление, переменная `i` будет иметь значение 10, и все 10 замыканий будут совместно использовать это значение. Таким образом, все функции в возвращаемом массиве будут возвращать одно и то же значение, что совсем не то, чего мы пытались добиться. Важно помнить, что цепочка областей видимости, связанная с замыканием, не фиксируется. Вложенные функции не создают частные копии области видимости и не фиксируют значения переменных.

Кроме того, при создании замыканий следует помнить, что `this` – это ключевое слово, а не переменная. Как отмечалось выше, каждый вызов функции получает свое значение `this`, и замыкание не имеет доступа к значению `this` внешней функции, если внешняя функция не сохранит его в переменной:

```
var self = this; // Сохранить значение this в переменной для использования
                 // во вложенной функции.
```

То же относится и к объекту `arguments`. Это не ключевое слово, но он автоматически создается при каждом вызове функции. Поскольку замыкания при вызове получают собственный объект `arguments`, они не могут обращаться к массиву аргументов внешней функции, если внешняя функция не сохранит этот массив в переменной с другим именем:

```
var outerArguments = arguments; // Сохранить для использования во вложенных функциях
```

В примере 8.5, далее в этой главе, определяется замыкание, использующее эти приемы для получения доступа к значениям `this` и `arguments` внешней функции.

8.7. Свойства и методы функций и конструктор Function

Мы видели, что в JavaScript-программах функции могут использоваться как значения. Оператор `typeof` возвращает для функций строку «function», однако в действительности функции в языке JavaScript – это особого рода объекты. А раз функции являются объектами, то они имеют свойства и методы, как любые другие объекты. Существует даже конструктор `Function()`, который создает новые объекты функций. В следующих подразделах описываются свойства и методы функций, а также конструктор `Function()`. Кроме того, информация обо всем этом приводится в справочном разделе.

8.7.1. Свойство length

В теле функции свойство `arguments.length` определяет количество аргументов, переданных функции. Однако свойство `length` самой функции имеет иной смысл. Это свойство, доступное только для чтения, возвращает количество аргументов, которое функция *ожидает* получить, – число объявленных параметров.

В следующем фрагменте определяется функция с именем `check()`, получающая массив аргументов `arguments` от другой функции. Она сравнивает свойство `arguments.length` (число фактически переданных аргументов) со свойством `arguments.callee.length` (число ожидаемых аргументов), чтобы определить, передано ли функции столько аргументов, сколько она ожидает. Если значения не совпадают, генерируется исключение. За функцией `check()` следует тестовая функция `f()`, демонстрирующая порядок использования функции `check()`:

```
// Эта функция использует arguments.callee, поэтому она
// не будет работать в строгом режиме.
function check(args) {
    var actual = args.length;           // Фактическое число аргументов
    var expected = args.callee.length;  // Ожидаемое число аргументов
    if (actual !== expected)            // Если не совпадают, генерируется исключение
        throw new Error("ожидается: " + expected + "; получено " + actual);
}

function f(x, y, z) {
    // Проверить число ожидаемых и фактически переданных аргументов.
    check(arguments);
    // Теперь выполнить оставшуюся часть функции как обычно
    return x + y + z;
}
```

8.7.2. Свойство prototype

Любая функция имеет свойство `prototype`, ссылающееся на объект, известный как *объект прототипа*. Каждая функция имеет свой объект прототипа. Когда функция используется в роли конструктора, вновь созданный объект наследует

свойства этого объекта прототипа. Прототипы и свойство `prototype` обсуждались в разделе 6.1.3, и мы еще раз вернемся к этим понятиям в главе 9.

8.7.3. Методы call() и apply()

Методы `call()` и `apply()` позволяют выполнять косвенный вызов функции (раздел 8.2.4), как если бы она была методом некоторого другого объекта. (Мы уже использовали метод `call()` в примере 6.4 для вызова `Object.prototype.toString` относительно объекта, класс которого необходимо было определить.) Первым аргументом обоим методам, `call()` и `apply()`, передается объект, относительно которого вызывается функция; этот аргумент определяет контекст вызова и становится значением ключевого слова `this` в теле функции. Чтобы вызвать функцию `f()` (без аргументов) как метод объекта `o`, можно использовать любой из методов, `call()` или `apply()`:

```
f.call(o);
f.apply(o);
```

Любой из этих способов вызова эквивалентен следующему фрагменту (где предполагается, что объект `o` не имеет свойства с именем `m`):

```
o.m = f;     // Временно сделать f методом o.
o.m();       // Вызывать его без аргументов.
delete o.m; // Удалить временный метод.
```

В строгом режиме ECMAScript 5 первый аргумент методов `call()` и `apply()` становится значением `this`, даже если это простое значение, `null` или `undefined`. В ECMAScript 3 и в нестрогом режиме значения `null` и `undefined` замещаются глобальным объектом, а простое значение – соответствующим объектом-оберткой.

Все остальные аргументы метода `call()`, следующие за первым аргументом, определяющим контекст вызова, передаются вызываемой функции. Например, ниже показано, как можно передать функции `f()` два числа и вызвать ее, как если бы она была методом объекта `o`:

```
f.call(o, 1, 2);
```

Метод `apply()` действует подобно методу `call()`, за исключением того, что аргументы для функции передаются в виде массива:

```
f.apply(o, [1,2]);
```

Если функция способна обрабатывать произвольное число аргументов, метод `apply()` может использоваться для вызова такой функции в контексте массива произвольной длины. Например, чтобы отыскать наибольшее число в массиве чисел, для передачи элементов массива функции `Math.max()` можно было бы использовать метод `apply()`:

```
var biggest = Math.max.apply(Math, array_of_numbers);
```

Обратите внимание, что метод `apply()` может работать не только с настоящими массивами, но и с объектами, подобными массивам. В частности, вы можете вызвать функцию с теми же аргументами, что и текущую функцию, передав массив с аргументами непосредственно методу `apply()`. Этот прием демонстрируется ниже:

```
// Замещает метод m объекта o версией метода, которая регистрирует
// сообщения до и после вызова оригинального метода.
```

```
function trace(o, m) {
    var original = o[m];          // Сохранить оригинальный метод в замыкании.
    o[m] = function() {           // Определить новый метод.
        console.log(new Date(), "Entering:", m);      // Записать сообщение.
        var result = original.apply(this, arguments); // Вызвать оригинал.
        console.log(new Date(), "Exiting:", m);       // Записать сообщение.
        return result;                                // Вернуть результат.
    };
}
```

Эта функция `trace()` принимает объект и имя метода. Она замещает указанный метод новым методом, который «обертывает» оригинальный метод дополнительной функциональностью. Такой прием динамического изменения существующих методов иногда называется «обезьяньей заплатой» («monkey-patching»).

8.7.4. Метод bind()

Метод `bind()` впервые появился в ECMAScript 5, но его легко имитировать в ECMAScript 3. Как следует из его имени, основное назначение метода `bind()` состоит в том, чтобы связать (bind) функцию с объектом. Если вызвать метод `bind()` функции `f` и передать ему объект `o`, он вернет новую функцию. Вызов новой функции (как обычной функции) выполнит вызов оригинальной функции `f` как метода объекта `o`. Любые аргументы, переданные новой функции, будут переданы оригинальной функции. Например:

```
function f(y) { return this.x + y; } // Функция, которую требуется привязать
var o = { x : 1 };                   // Объект, к которому выполняется привязка
var g = f.bind(o);                   // Вызов g(x) вызовет o.f(x)
g(2)                                 // => 3
```

Такой способ связывания легко реализовать в ECMAScript 3, как показано ниже:

```
// Возвращает функцию, которая вызывает f как метод объекта o
// и передает ей все свои аргументы.
function bind(f, o) {
    if (f.bind) return f.bind(o);     // Использовать метод bind, если имеется
    else return function() {          // Иначе связать, как показано ниже
        return f.apply(o, arguments);
    };
}
```

Метод `bind()` в ECMAScript 5 не просто связывает функцию с объектом. Он также выполняет частичное применение: помимо значения `this` связаны будут все аргументы, переданные методу `bind()` после первого его аргумента. Частичное применение – распространенный прием в функциональном программировании и иногда называется *каррингом* (*currying*). Ниже приводится несколько примеров использования метода `bind()` для частичного применения:

```
var sum = function(x,y) { return x + y };  // Возвращает сумму 2 аргументов
// Создать новую функцию, подобную sum, но со связанным значением null
// ключевого слова this и со связанным значением первого аргумента, равным 1.
// Новая функция принимает всего один аргумент.
var succ = sum.bind(null, 1);
succ(2) // => 3: аргумент x связан со значением 1, а 2 передается в арг. y
```

```
function f(y,z) { return this.x + y + z }; // Еще одна функция сложения
var g = f.bind({x:1}, 2);                  // Связать this и y
g(3) // => 6: this.x - связан с 1, y - связан с 2, а 3 передается в z
```

В ECMAScript 3 также возможно связывать значение this и выполнять частичное применение. Стандартный метод bind() можно имитировать программным кодом, который приводится в примере 8.5. Обратите внимание, что этот метод сохраняется как Function.prototype.bind, благодаря чему все функции наследуют его. Данный прием подробно рассматривается в разделе 9.4.

Пример 8.5. Метод Function.bind() для ECMAScript 3

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, аргументы */) {
        // Сохранить this и arguments в переменных, чтобы их можно было
        // использовать во вложенной функции ниже.
        var self = this, boundArgs = arguments;

        // Возвращаемое значение метода bind() - функция
        return function() {
            // Сконструировать список аргументов, начиная со второго аргумента
            // метода bind, и передать все эти аргументы указанной функции.
            var args = [], i;
            for(i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]);
            for(i = 0; i < arguments.length; i++) args.push(arguments[i]);

            // Теперь вызвать self как метод объекта o со всеми аргументами
            return self.apply(o, args);
        };
    };
}
```

Обратите внимание, что функция, возвращаемая этим методом bind(), является замыканием, использующим переменные self и boundArgs, объявленные во внешней функции, которые остаются доступными вложенной функции даже после того, как она будет возвращена внешней функцией и вызвана из-за пределов внешней функции.

Метод bind(), определяемый стандартом ECMAScript 5, имеет некоторые особенности, которые невозможно реализовать в ECMAScript 3. Прежде всего, настоящий метод bind() возвращает объект функции, свойство length которой установлено в соответствии с количеством параметров связываемой функции, минус количество связанных аргументов (но не меньше нуля). Во-вторых, метод bind() в ECMAScript 5 может использоваться для частичного применения функций-конструкторов. Если функцию, возвращаемую методом bind(), использовать как конструктор, значение this, переданное методу bind(), игнорируется, и оригинальная функция будет вызвана как конструктор, с уже связанными аргументами, если они были определены. Функции, возвращаемые методом bind(), не имеют свойства prototype (свойство prototype обычных функций нельзя удалить), и объекты, созданные связанными функциями-конструкторами, наследуют свойство prototype оригинального, несвязанного конструктора. Кроме того, с точки зрения оператора instanceof связанные конструкторы действуют точно так же, как несвязанные конструкторы.

8.7.5. Метод toString()

Подобно другим объектам в языке JavaScript, функции имеют метод `toString()`. Спецификация ECMAScript требует, чтобы этот метод возвращал строку, следующую синтаксису инструкции объявления функции. На практике большинство (но не все) реализаций метода `toString()` возвращают полный исходный текст функции. Для встроенных функций обычно возвращается строка, содержащая вместо тела функции текст «[native code]» или аналогичный.

8.7.6. Конструктор Function()

Функции обычно определяются с помощью ключевого слова `function` либо в форме инструкции объявления функции, либо в форме выражения-литерала. Однако функции могут также определяться с помощью конструктора `Function()`. Например:

```
var f = new Function("x", "y", "return x*y;");
```

Эта строка создаст новую функцию, которая более или менее эквивалентна функции, объявленной с помощью более привычного синтаксиса:

```
var f = function(x, y) { return x*y; }
```

Конструктор `Function()` принимает произвольное число строковых аргументов. Последний аргумент должен содержать текст с телом функции; он может включать произвольное число инструкций на языке JavaScript, разделенных точкой с запятой. Все остальные аргументы конструктора интерпретируются как имена параметров функции. Чтобы создать функцию, не имеющую аргументов, достаточно передать конструктору всего одну строку – тело функции.

Примечательно, что конструктору `Function()` не передается никаких аргументов, определяющих имя создаваемой функции. Подобно литералам функций, конструктор `Function()` создает анонимные функции.

Есть несколько моментов, связанных с конструктором `Function()`, о которых следует упомянуть особо:

- Конструктор `Function()` позволяет динамически создавать и компилировать функции в процессе выполнения программы.
- При каждом вызове конструктор `Function()` выполняет синтаксический анализ тела функции и создает новый объект функции. Если вызов конструктора производится в теле цикла или часто вызываемой функции, это может отрицательно сказаться на производительности программы. Напротив, вложенные функции и выражения определения функций внутри циклов не компилируются повторно.
- И последний, очень важный момент: когда функция создается с помощью конструктора `Function()`, не учитывается лексическая область видимости – функции всегда компилируются как глобальные функции, что наглядно демонстрирует следующий фрагмент:

```
var scope = "глобальная";
function constructFunction() {
    var scope = "локальная";
```

```
        return new Function("return scope"); // Здесь не используется
                                             // локальная область видимости!
    }
    // Следующая строка вернет "глобальная", потому что функция, возвращаемая
    // конструктором Function(), является глобальной.
    constructFunction()();         // => "глобальная"
```

Точнее всего конструктор `Function()` соответствует глобальной версии `eval()` (раздел 4.12.2), которая определяет новые переменные и функции в своей собственной области видимости. Вам редко придется использовать этот конструктор в своих программах.

8.7.7. Вызываемые объекты

В разделе 7.11 мы узнали, что существуют объекты, «подобные массивам», которые не являются настоящими массивами, но во многих случаях могут интерпретироваться как массивы. Аналогичная ситуация складывается с функциями. *Вызываемый объект* – это любой объект, который может быть вызван в выражении вызова функции. Все функции являются вызываемыми объектами, но не все вызываемые объекты являются функциями.

Вызываемые объекты, не являющиеся функциями, встречаются в современных реализациях JavaScript в двух ситуациях. Во-первых, веб-броузер IE (версии 8 и ниже) реализует клиентские методы, такие как `Window.alert()` и `Document.getElementsById()`, используя вызываемые объекты, а не объекты класса `Function`. Эти методы действуют в IE точно так же, как в других броузерах, но они не являются объектами `Function`. В IE9 был выполнен переход на использование настоящих функций, поэтому со временем эта разновидность вызываемых объектов будет использоваться все меньше и меньше.

Другой типичной разновидностью вызываемых объектов являются объекты `RegExp` – во многих броузерах предоставляется возможность напрямую вызывать объект `RegExp`, как более краткий способ вызова его метода `exec()`. Эта возможность не предусматривается стандартом JavaScript. В свое время она была реализована компанией Netscape и подхвачена другими производителями для обеспечения совместимости. Старайтесь не писать программы, опирающиеся на возможность вызова объектов `RegExp`: данная особенность, скорее всего, будет объявлена нерекомендуемой и будет ликвидирована в будущем. Оператор `typeof` не во всех броузерах одинаково распознает вызываемые объекты `RegExp`. В одних броузерах он возвращает строку «function», а в других – «object».

Если в программе потребуется определить, является ли объект настоящим объектом функции (и обладает методами функций), сделать это можно, определив значение атрибута `class` (раздел 6.8.2), использовав прием, продемонстрированный в примере 6.4:

```
function isFunction(x) {
    return Object.prototype.toString.call(x) === "[object Function]";
}
```

Обратите внимание, насколько эта функция `isFunction()` похожа на функцию `isArray()`, представленную в разделе 7.10.

8.8. Функциональное программирование

JavaScript не является языком функционального программирования, как Lisp или Haskell, но тот факт, что программы на языке JavaScript могут манипулировать функциями как объектами означает, что в JavaScript можно использовать приемы функционального программирования. Масса методов в ECMAScript 5, таких как `map()` и `reduce()`, сами по себе способствуют использованию функционального стиля программирования. В следующих разделах демонстрируются приемы функционального программирования на языке JavaScript. Их цель – не подтолкнуть вас к использованию этого замечательного стиля программирования, а показать широту возможностей функций в языке JavaScript.[1]

8.8.1. Обработка массивов с помощью функций

Представим, что у нас имеется массив чисел и нам необходимо найти среднее значение и стандартное отклонение для этих значений. Эту задачу можно было бы решить без использования приемов функционального программирования, как показано ниже:

```
var data = [1,1,3,5,5];         // Массив чисел

// Среднее - это сумма значений элементов, деленная на их количество
var total = 0;
for(var i = 0; i < data.length; i++) total += data[i];
var mean = total/data.length;  // Среднее значение равно 3

// Чтобы найти стандартное отклонение, необходимо вычислить сумму квадратов
// отклонений элементов от среднего.
total = 0;
for(var i = 0; i < data.length; i++) {
    var deviation = data[i] - mean;
    total += deviation * deviation;
}
var stddev = Math.sqrt(total/(data.length-1)); // Стандартное отклонение = 2
```

Те же вычисления можно выполнить в более кратком функциональном стиле, задействовав методы массивов `map()` и `reduce()`, как показано ниже (краткое описание этих методов приводится в разделе 7.9):

```
// Для начала необходимо определить две простые функции
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };

// Затем использовать их совместно с методами класса Array для вычисления
// среднего и стандартного отклонения
var data = [1,1,3,5,5];
var mean = data.reduce(sum)/data.length;
var deviations = data.map(function(x) {return x-mean;});
var stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
```

---

[1] Если эта тема вам любопытна, вероятно, вас заинтересует возможность использования (или хотя бы знакомства) библиотеки Functional JavaScript Оливера Стила (Oliver Steel), которую можно найти по адресу: *http://osteele.com/sources/javascript/functional/*.

А как быть, если в нашем распоряжении имеется только реализация ECMAScript 3, где отсутствуют эти новейшие методы массивов? Можно определить собственные функции `map()` и `reduce()`, которые будут использовать встроенные методы при их наличии:

```
// Вызывает функцию f для каждого элемента массива и возвращает массив результатов.
// Использует метод Array.prototype.map, если он определен.
var map = Array.prototype.map
    ? function(a, f) { return a.map(f); } // Если метод map() доступен
    : function(a,f) {                     // Иначе реализовать свою версию
        var results = [];
        for(var i = 0, len = a.length; i < len; i++) {
            if (i in a) results[i] = f.call(null, a[i], i, a);
        }
        return results;
};

// Выполняет свертку массива в единственное значение, используя функцию f
// и необязательное начальное значение. Использует метод Array.prototype.reduce,
// если он определен.
var reduce = Array.prototype.reduce
    ? function(a, f, initial) {           // Если метод reduce() доступен.
        if (arguments.length > 2)
            return a.reduce(f, initial);  // Если указано начальное значение.
        else return a.reduce(f);          // Иначе без начального значения.
    }
    : function(a, f, initial) {           // Этот алгоритм взят из спецификации ES5
        var i = 0, len = a.length, accumulator;

        // Использовать указанное начальное значение или первый элемент a
        if (arguments.length > 2) accumulator = initial;
        else { // Найти первый элемент массива с определенным значением
            if (len == 0) throw TypeError();
            while(i < len) {
                if (i in a) {
                    accumulator = a[i++];
                    break;
                }
                else i++;
            }
            if (i == len) throw TypeError();
        }

        // Теперь вызвать f для каждого оставшегося элемента массива
        while(i < len) {
            if (i in a)
                accumulator = f.call(undefined, accumulator, a[i], i, a);
            i++;
        }

        return accumulator;
    };
```

После определения этих функций `map()` и `reduce()` вычисление среднего и стандартного отклонения будет выглядеть так:

```
var data = [1,1,3,5,5];
var sum = function(x,y) { return x+y; };
var square = function(x) { return x*x; };
var mean = reduce(data, sum)/data.length;
var deviations = map(data, function(x) {return x-mean;});
var stddev = Math.sqrt(reduce(map(deviations, square), sum)/(data.length-1));
```

8.8.2. Функции высшего порядка

Функции высшего порядка – это функции, которые оперируют функциями, принимая одну или более функций и возвращая новую функцию. Например:

```
// Эта функция высшего порядка возвращает новую функцию, которая передает свои аргументы
// функции f и возвращает логическое отрицание значения, возвращаемого функцией f;
function not(f) {
    return function() {                          // Возвращает новую функцию
        var result = f.apply(this, arguments); // вызов f
        return !result;                          // и инверсия результата.
    };
}

var even = function(x) { // Функция, определяющая четность числа
    return x % 2 === 0;
};

var odd = not(even);       // Новая функция, выполняющая противоположную операцию
[1,1,3,5,5].every(odd);  // => true: все элементы массива нечетные
```

Функция `not()` в примере выше является функцией высшего порядка, потому что она принимает функцию в виде аргумента и возвращает новую функцию. В качестве еще одного примера рассмотрим функцию `mapper()`, представленную ниже. Она принимает функцию в виде аргумента и возвращает новую функцию, которая отображает один массив в другой, применяя указанную функцию. Данная функция использует функцию `map()`, которая была определена выше, и важно понимать, чем отличаются эти две функции:

```
// Возвращает функцию, которая принимает массив в виде аргумента, применяет функцию f
// к каждому элементу и возвращает массив возвращаемых значений.
// Эта функция отличается от функции map(), представленной выше.
function mapper(f) {
    return function(a) { return map(a, f); };
}

var increment = function(x) { return x+1; };
var incrementer = mapper(increment);
incrementer([1,2,3]) // => [2,3,4]
```

Ниже приводится пример еще одной, более универсальной функции, которая принимает две функции, `f` и `g`, и возвращает новую функцию, которая возвращает результат `f(g())`:

```
// Возвращает новую функцию, которая вычисляет f(g(...)). Возвращаемая функция h
// передает все свои аргументы функции g, затем передает значение, полученное от g,
// функции f и возвращает результат вызова f. Обе функции, f и g,
// вызываются с тем же значением this, что и h.
function compose(f,g) {
    return function() {
        // Для вызова f используется call, потому что ей передается
        // единственное значение, а для вызова g используется apply,
        // потому что ей передается массив значений.
        return f.call(this, g.apply(this, arguments));
    };
}
var square = function(x) { return x*x; };
var sum = function(x,y) { return x+y; };
var squareofsum = compose(square, sum);
squareofsum(2,3)                    // => 25
```

Функции `partial()` и `memoize()`, которые определяются в следующем разделе, представляют собой еще две важные функции высшего порядка.

8.8.3. Частичное применение функций

Метод `bind()` функции `f` (раздел 8.7.4) возвращает новую функцию, которая вызывает `f` в указанном контексте и с заданным набором аргументов. Можно сказать, что он связывает функцию с объектом контекста и частично применяет аргументы. Метод `bind()` применяет аргументы слева, т. е. аргументы, которые передаются методу `bind()`, помещаются в начало списка аргументов, передаваемых оригинальной функции. Однако есть возможность частичного применения аргументов справа:

```
// Вспомогательная функция преобразования объекта (или его части),
// подобного массиву, в настоящий массив. Используется ниже
// для преобразования объекта arguments в настоящий массив.
function array(a, n) { return Array.prototype.slice.call(a, n || 0); }

// Аргументы этой функции помещаются в начало списка
function partialLeft(f /*, ...*/) {
    var args = arguments;              // Сохранить внешний массив аргументов
    return function() {                // И вернуть эту функцию
        var a = array(args, 1);        // Начиная с элемента 1 во внеш. масс.
        a = a.concat(array(arguments)); // Добавить внутренний массив аргум.
        return f.apply(this, a);       // Вызвать f с этим списком аргументов
    };
}

// Аргументы этой функции помещаются в конец списка
function partialRight(f /*, ...*/) {
    var args = arguments;              // Сохранить внешний массив аргументов
    return function() {                // И вернуть эту функцию
        var a = array(arguments);      // Начинать с внутр. масс. аргументов
        a = a.concat(array(args,1));   // Добавить внешние арг., начиная с 1.
        return f.apply(this, a);       // Вызвать f с этим списком аргументов
    };
}

// Аргументы этой функции играют роль шаблона. Неопределенные значения
// в списке аргументов заполняются значениями из внутреннего набора.
function partial(f /*, ... */) {
    var args = arguments;              // Сохранить внешний массив аргументов
    return function() {
        var a = array(args, 1);        // Начинать с внешнего массива аргументов
        var i=0, j=0;
        // Цикл по этим аргументам, заменить значения undefined значениями
        // из внутреннего списка аргументов
        for(; i < a.length; i++)
            if (a[i] === undefined) a[i] = arguments[j++];
        // Добавить оставшиеся внутренние аргументы в конец списка
        a = a.concat(array(arguments, j))
        return f.apply(this, a);
    };
}

// Ниже приводится функция, принимающая три аргумента
var f = function(x,y,z) { return x * (y - z); };
// Обратите внимание на отличия между следующими тремя частичными применениями
partialLeft(f, 2)(3,4)        // => -2: Свяжет первый аргумент:    2 * (3 - 4)
partialRight(f, 2)(3,4)       // =>  6: Свяжет последний аргумент: 3 * (4 - 2)
partial(f, undefined, 2)(3,4) // => -6: Свяжет средний аргумент:   3 * (2 - 4)
```

Эти функции частичного применения позволяют легко объявлять новые функции на основе уже имеющихся функций. Например:

```
var increment = partialLeft(sum, 1);
var cuberoot = partialRight(Math.pow, 1/3);
String.prototype.first = partial(String.prototype.charAt, 0);
String.prototype.last = partial(String.prototype.substr, -1, 1);
```

Прием частичного применения становится еще более интересным, когда он используется в комбинации с функциями высшего порядка. Например, ниже демонстрируется еще один способ определения функции `not()`, представленной выше, за счет совместного использования приемов композиции и частичного применения:

```
var not = partialLeft(compose, function(x) { return !x; });
var even = function(x) { return x % 2 === 0; };
var odd = not(even);
var isNumber = not(isNaN)
```

Прием композиции и частичного применения можно также использовать для вычисления среднего значения и стандартного отклонения в крайне функциональном стиле:

```
var data = [1,1,3,5,5];                      // Исходные данные
var sum = function(x,y) { return x+y; };     // Две элементарные функции
var product = function(x,y) { return x*y; };
var neg = partial(product, -1);              // Определения других функций
var square = partial(Math.pow, undefined, 2);
var sqrt = partial(Math.pow, undefined, .5);
var reciprocal = partial(Math.pow, undefined, -1);

// Вычислить среднее и стандартное отклонение. Далее используются только функции
// без каких либо операторов, отчего программный код начинает напоминать
// программный код на языке Lisp!
var mean = product(reduce(data, sum), reciprocal(data.length));
var stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))),
                                 sum),
                          reciprocal(sum(data.length,-1))));
```

8.8.4. Мемоизация

В разделе 8.4.1 была определена функция нахождения факториала, которая сохраняет ранее вычисленные результаты. В функциональном программировании такого рода кэширование называется *мемоизацией* (*memorization*). В следующем примере демонстрируется функция `memoize()` высшего порядка, которая принимает функцию в виде аргумента и возвращает ее мемоизованную версию:

```
// Возвращает мемоизованную версию функции f. Работает, только если все возможные
// аргументы f имеют отличающиеся строковые представления.
function memoize(f) {
    var cache = {}; // Кэш значений сохраняется в замыкании.

    return function() {
        // Создать строковую версию массива arguments для использования
        // в качестве ключа кэша.
        var key = arguments.length + Array.prototype.join.call(arguments,",");
        if (key in cache) return cache[key];
        else return cache[key] = f.apply(this, arguments);
    };
}
```

Функция `memoize()` создает новый объект для использования в качестве кэша и присваивает его локальной переменной, благодаря чему он остается доступным (через замыкание) только для возвращаемой функции. Возвращаемая функция преобразует свой массив `arguments` в строку и использует ее как имя свойства объекта-кэша. Если значение присутствует в кэше, оно просто возвращается в качестве результата. В противном случае вызывается оригинальная функция, вычисляющая значение для заданной комбинации значений аргументов; полученное значение помещается в кэш и возвращается. Следующий фрагмент демонстрирует, как можно использовать функцию `memoize()`:

```
// Возвращает наибольший общий делитель двух целых чисел, используя
// алгоритм Эвклида: http://en.wikipedia.org/wiki/Euclidean_algorithm
function gcd(a,b) {                  // Проверка типов a и b опущена
    var t;                           // Временная переменная для обмена
    if (a < b) t=b, b=a, a=t;        // Убедиться, что a >= b
    while(b != 0) t=b, b = a%b, a=t; // Это алгоритм Эвклида поиска НОД
    return a;
}

var gcdmemo = memoize(gcd);
gcdmemo(85, 187) // => 17

// Обратите внимание, что при мемоизации рекурсивных функций желательно,
// чтобы рекурсия выполнялась в мемоизованной версии, а не в оригинале.
var factorial = memoize(function(n) {
                            return (n <= 1) ? 1 : n * factorial(n-1);
                       });
factorial(5) // => 120. Также поместит в кэш факториалы для чисел 4, 3, 2 и 1.
```

9

Классы и модули

Введение в JavaScript-объекты было дано в главе 6, где каждый объект трактовался как уникальный набор свойств, отличающих его от любых других объектов. Однако часто бывает полезнее определить *класс* объектов, обладающих общими свойствами. Члены, или *экземпляры*, класса обладают собственными свойствами, определяющими их состояние, но они также обладают свойствами (обычно методами), определяющими их поведение. Эти особенности поведения определяются классом и являются общими для всех экземпляров. Например, можно объявить класс `Complex` для представления комплексных чисел и выполнения арифметических операций с ними. Экземпляр класса `Complex` мог бы обладать свойствами для хранения действительной и мнимой частей комплексного числа. А класс `Complex` мог бы определять методы, выполняющие операции сложения и умножения (поведение) этих чисел.

Классы в языке JavaScript основаны на использовании механизма наследования прототипов. Если два объекта наследуют свойства от одного и того же объекта-прототипа, говорят, что они принадлежат одному классу. С прототипами и наследованием мы познакомились в разделах 6.1.3 и 6.2.2. Сведения из этих разделов вам обязательно потребуются для понимания того, о чем рассказывается в этой главе. В этой главе прототипы будут рассматриваться в разделе 9.1.

Если два объекта наследуют один и тот же прототип, обычно (но не обязательно) это означает, что они были созданы и инициализированы с помощью одного конструктора. С конструкторами мы познакомились в разделах 4.6, 6.1.2 и 8.2.3. Дополнительные сведения о них в этой главе приводятся в разделе 9.2.

Те, кто знаком со строго типизированными объектно-ориентированными языками программирования, такими как Java или C++, могут заметить, что классы в языке JavaScript совершенно не похожи на классы в этих языках. Конечно, есть некоторые синтаксические сходства, и имеется возможность имитировать многие особенности «классических» классов в JavaScript. Но лучше будет с самого начала понять, что классы и механизм наследования на основе прототипов в языке JavaScript существенно отличаются от классов и механизма наследования на основе классов в языке Java и подобных ему. В разделе 9.3 демонстрируются приемы имитации классических классов на языке JavaScript.

Еще одной важной особенностью классов в языке JavaScript является возможность динамического расширения. Эта особенность описывается в разделе 9.4. Классы можно также интерпретировать как типы данных, и в разделе 9.5 будет представлено несколько способов определения класса объекта. В этом разделе вы также познакомитесь с философией программирования, известной как «утиная типизация» («duck-typing»), которая во главу угла ставит не тип объекта, а его возможности.

После знакомства со всеми этими основами объектно-ориентированного программирования в JavaScript мы перейдем в этой же главе к изучению более практического материала. В разделе 9.6 будут представлены два примера непростых классов и продемонстрировано несколько практических объектно-ориентированных приемов расширения этих классов. В разделе 9.7 будет показано (на множестве примеров), как расширять или наследовать другие классы и как создавать иерархии классов в языке JavaScript. В разделе 9.8 рассматриваются дополнительные приемы работы с классами с использованием новых возможностей, появившихся в ECMAScript 5.

Определение классов – это один из способов создания модульного программного кода многократного использования. В последнем разделе этой главы мы в общих чертах поговорим о модулях в языке JavaScript.

9.1. Классы и прототипы

В языке JavaScript класс – это множество объектов, наследующих свойства от общего объекта-прототипа. Таким образом, объект-прототип является центральной особенностью класса. В примере 6.1 была определена функция `inherit()`, возвращающая вновь созданный объект, наследующий указанный объект-прототип. Если определить объект-прототип и затем воспользоваться функцией `inherit()` для создания объектов, наследующих его, фактически будет создан класс JavaScript. Обычно экземпляры класса требуют дополнительной инициализации, поэтому обычно определяется функция, которая создает и инициализирует новые объекты. В примере 9.1 демонстрируется такая функция: она определяет объект-прототип класса, представляющего диапазон значений, а также «фабричную» функцию, которая создает и инициализирует новые экземпляры класса.

Пример 9.1. Простой класс JavaScript

```
// range.js: Класс, представляющий диапазон значений.
// Это фабричная функция, которая возвращает новый объект range.
function range(from, to) {
    // Использует функцию inherit() для создания объекта, наследующего объект-прототип,
    // определяемый ниже. Объект-прототип хранится как свойство данной функции
    // и определяет общие методы (поведение) для всех объектов range.
    var r = inherit(range.methods);

    // Сохранить начальное и конечное значения в новом объекте range.
    // Это не унаследованные свойства, и они являются уникальными для данного объекта.
    r.from = from;
    r.to = to;
    // В заключение вернуть новый объект
    return r;
}

// Ниже следует объект-прототип, определяющий методы, наследуемые всеми объектами range.
range.methods = {
    // Возвращает true, если x - объект класса range, в противном случае возвращает false
    // Этот метод может работать не только с числовыми диапазонами,
    // но также с текстовыми диапазонами и с диапазонами дат Date.
    includes: function(x) { return this.from <= x && x <= this.to; },
    // Вызывает f для каждого целого числа в диапазоне.
    // Этот метод может работать только с числовыми диапазонами.
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // Возвращает строковое представление диапазона
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Ниже приводится пример использования объекта range.
var r = range(1,3);         // Создать новый объект range
r.includes(2);              // => true: число 2 входит в диапазон
r.foreach(console.log);     // Выведет 1 2 3
console.log(r);             // Выведет (1...3)
```

В примере 9.1 есть несколько интересных моментов, которые следует отметить особо. Здесь определяется фабричная функция `range()`, которая используется для создания новых объектов `range`. Обратите внимание, что для хранения объекта-прототипа, определяющего класс, используется свойство `range.methods` функции `range()`. В таком способе хранения объекта-прототипа нет ничего необычного. Во-вторых, отметьте, что функция `range()` определяет свойства `from` и `to` для каждого объекта `range`. Эти не общие, не унаследованные свойства определяют уникальную информацию для каждого отдельного объекта `range`. Наконец, обратите внимание, что все общие, унаследованные методы, определяемые свойством `range.methods`, используют свойства `from` и `to` и ссылаются на них с помощью ключевого слова `this`, указывающего на объект, относительно которого вызываются эти методы. Такой способ использования `this` является фундаментальной характеристикой методов любого класса.

9.2. Классы и конструкторы

В примере 9.1 демонстрируется один из способов определения класса в языке JavaScript. Однако это не самый типичный способ, потому что он не связан с определением *конструктора*. Конструктор – это функция, предназначенная для инициализации вновь созданных объектов. Как описывалось в разделе 8.2.3, конструкторы вызываются с помощью ключевого слова `new`. Применение ключевого слова `new` при вызове конструктора автоматически создает новый объект, поэтому конструктору остается только инициализировать свойства этого нового объекта. Важной особенностью вызова конструктора является использование свойства `prototype` конструктора в качестве прототипа нового объекта. Это означает, что все объекты, созданные с помощью одного конструктора, наследуют один и тот же объект-прототип и, соответственно, являются членами одного и того же класса. В примере 9.2 демонстрируется, как можно было бы реализовать класс range, представленный в примере 9.1, не с помощью фабричной функции, а с помощью функции конструктора:

Пример 9.2. Реализация класса Range с помощью конструктора

```
// range2.js: Еще один класс, представляющий диапазон значений.
// Это функция-конструктор, которая инициализирует новые объекты Range.
// Обратите внимание, что она не создает и не возвращает объект.
// Она лишь инициализирует его.
function Range(from, to) {
    // Сохранить начальное и конечное значения в новом объекте range.
    // Это не унаследованные свойства, и они являются уникальными для данного объекта.
    this.from = from;
    this.to = to;
}

// Все объекты Range наследуют свойства этого объекта.
// Обратите внимание, что свойство обязательно должно иметь имя "prototype".
Range.prototype = {
    // Возвращает true, если x - объект класса range, в противном случае возвращает false
    // Этот метод может работать не только с числовыми диапазонами, но также
    // с текстовыми диапазонами и с диапазонами дат Date.
    includes: function(x) { return this.from <= x && x <= this.to; },
    // Вызывает f для каждого целого числа в диапазоне.
    // Этот метод может работать только с числовыми диапазонами.
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    // Возвращает строковое представление диапазона
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Ниже приводится пример использования объекта range.
var r = new Range(1,3);      // Создать новый объект range
r.includes(2);               // => true: число 2 входит в диапазон
r.foreach(console.log);      // Выведет 1 2 3
console.log(r);              // Выведет (1...3)
```

Имеет смысл детально сравнить примеры 9.1 и 9.2 и отметить различия между этими двумя способами определения классов. Во-первых, обратите внимание, что при преобразовании в конструктор фабричная функция range() была переименована в Range(). Это обычное соглашение по оформлению: функции-конструкторы в некотором смысле определяют классы, а имена классов начинаются с заглавных символов. Имена обычных функций и методов начинаются со строчных символов.

Далее отметьте, что конструктор Range() вызывается (в конце примера) с ключевым словом new, тогда как фабричная функция range() вызывается без него. В примере 9.1 для создания нового объекта использовался вызов обычной функции (раздел 8.2.1), а в примере 9.2 – вызов конструктора (раздел 8.2.3). Поскольку конструктор Range() вызывается с ключевым словом new, отпадает необходимость вызывать функцию inherit() или предпринимать какие-либо другие действия по созданию нового объекта. Новый объект создается автоматически перед вызовом конструктора и доступен в конструкторе как значение `this`. Конструктору `Range()` остается лишь инициализировать его. Конструкторы даже не должны возвращать вновь созданный объект. Выражение вызова конструктора автоматически создает новый объект, вызывает конструктор как метод этого объекта и возвращает объект. Тот факт, что вызов конструктора настолько отличается от вызова обычной функции, является еще одной причиной, почему конструкторам принято давать имена, начинающиеся с заглавного символа. Конструкторы предназначены для вызова в виде конструкторов, с ключевым словом `new`, и обычно при вызове в виде обычных функций они не способны корректно выполнять свою работу. Соглашение по именованию конструкторов, обеспечивающее визуальное отличие имен конструкторов от имен обычных функций, помогает программистам не забывать использовать ключевое слово `new`.

Еще одно важное отличие между примерами 9.1 и 9.2 заключается в способе именования объекта-прототипа. В первом примере прототипом было свойство `range.methods`. Это было удобное, описательное имя, но в значительной мере произвольное. Во втором примере прототипом является свойство `Range.prototype`, и это имя является обязательным. Выражение вызова конструктора `Range()` автоматически использует свойство `Range.prototype` как прототип нового объекта `Range`.

Наконец, обратите также внимание на одинаковые фрагменты примеров 9.1 и 9.2: в обоих классах методы объекта `range` определяются и вызываются одинаковым способом.

9.2.1. Конструкторы и идентификация класса

Как видите, объект-прототип играет чрезвычайно важную роль в идентификации класса: два объекта являются экземплярами одного класса, только если они наследуют один и тот же объект-прототип. Функция-конструктор, инициализирующая свойства нового объекта, не является определяющей: два конструктора могут иметь свойства `prototype`, ссылающиеся на один объект-прототип. В этом случае оба конструктора будут создавать экземпляры одного и того же класса.

Хотя конструкторы не играют такую же важную роль в идентификации класса, как прототипы, тем не менее конструкторы выступают в качестве фасада класса. Например, имя конструктора обычно используется в качестве имени класса. Так, принято говорить, что конструктор `Range()` создает объекты класса `Range`. Однако более важным применением конструкторов является их использование в операторе `instanceof` при проверке принадлежности объекта классу. Если имеется объект `r`, и необходимо проверить, является ли он объектом класса `Range`, такую проверку можно выполнить так:

```
r instanceof Range // вернет true, если r наследует Range.prototype
```

В действительности оператор `instanceof` не проверяет, был ли объект `r` инициализирован конструктором `Range`. Он проверяет, наследует ли этот объект свойство `Range.prototype`. Как бы то ни было, синтаксис оператора `instanceof` закрепляет использование конструкторов в качестве идентификаторов классов. Мы еще встретимся с оператором `instanceof` далее в этой главе.

9.2.2. Свойство constructor

В примере 9.2 свойству `Range.prototype` присваивался новый объект, содержащий методы класса. Хотя было удобно определить методы как свойства единственного объекта-литерала, но при этом совершенно не было необходимости создавать новый объект. Роль конструктора в языке JavaScript может играть любая функция, поскольку выражению вызова конструктора необходимо лишь свойство `prototype`. Следовательно, любая функция (кроме функций, возвращаемых методом `Function.bind()` в ECMAScript 5) автоматически получает свойство `prototype`. Значением этого свойства является объект, который имеет единственное неперечислимое свойство `constructor`. Значением свойства `constructor` является объект функции:

```
var F = function() {}; // Это объект функции.
var p = F.prototype;   // Это объект-прототип, связанный с ней.
var c = p.constructor; // Это функция, связанная с прототипом.
c === F                // => true: F.prototype.constructor === F для всех функций
```

Наличие предопределенного объекта-прототипа со свойством `constructor` означает, что объекты обычно наследуют свойство `constructor`, которое ссылается на их конструкторы. Поскольку конструкторы играют роль идентификаторов классов, свойство `constructor` определяет класс объекта:

```
var o = new F();    // Создать объект класса F
o.constructor === F // => true: свойство constructor определяет класс
```

Эти взаимосвязи между функцией-конструктором, ее прототипом, обратной ссылкой из прототипа на конструктор и экземплярами, созданными с помощью конструктора, иллюстрируются на рис. 9.1.

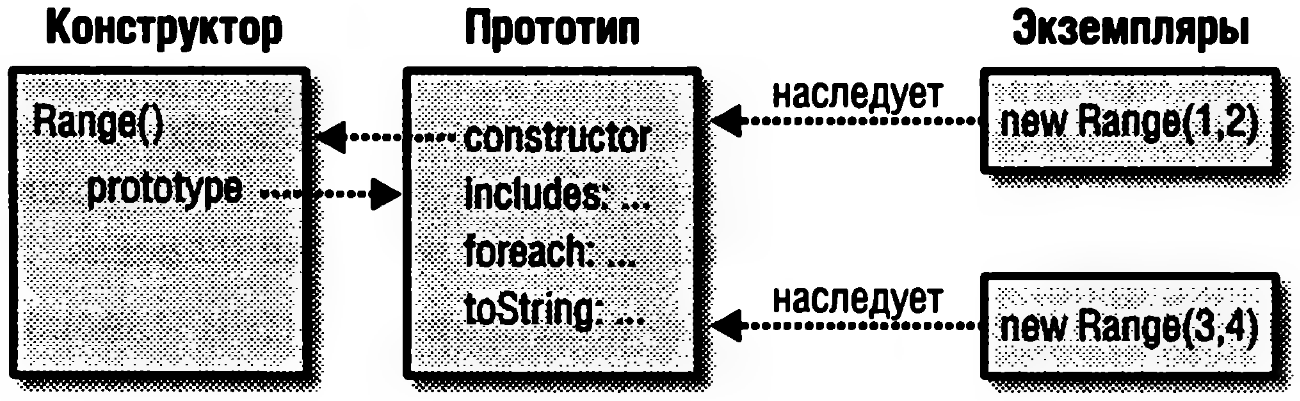


Рис. 9.1. Функция-конструктор, ее прототип и экземпляры

Обратите внимание, что в качестве примера для рис. 9.1 был взят наш конструктор `Range()`. Однако в действительности класс `Range`, определенный в примере 9.2, замещает предопределенный объект `Range.prototype` своим собственным. А новый объект-прототип не имеет свойства `constructor`. По этой причине экземпляры класса `Range`, как следует из определения, не имеют свойства `constructor`. Решить эту проблему можно, явно добавив конструктор в прототип:

```
Range.prototype = {
    constructor: Range, // Явно установить обратную ссылку на конструктор
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {
        for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
    },
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};
```

Другой типичный способ заключается в том, чтобы использовать предопределенный объект-прототип, который уже имеет свойство `constructor`, и добавлять методы в него:

```
// Здесь расширяется предопределенный объект Range.prototype,
// поэтому не требуется переопределять значение автоматически
// создаваемого свойства Range.prototype.constructor.
Range.prototype.includes = function(x) { return this.from<=x && x<=this.to; };
Range.prototype.foreach = function(f) {
for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
};
Range.prototype.toString = function() {
return "(" + this.from + "..." + this.to + ")";
};
```

9.3. Классы в стиле Java

Если вам приходилось программировать на языке Java или других объектно-ориентированных языках со строгим контролем типов, вы, возможно, привыкли считать, что классы могут иметь четыре типа *членов*:

Поля экземпляра

Это свойства, или переменные экземпляра, хранящие информацию о конкретном объекте.

Методы экземпляров

Методы, общие для всех экземпляров класса, которые вызываются относительно конкретного объекта.

Поля класса

Это свойства, или переменные, всего класса в целом, а не конкретного экземпляра.

Методы класса

Методы всего класса в целом, а не конкретного экземпляра.

Одна из особенностей языка JavaScript, отличающая его от языка Java, состоит в том, что функции в JavaScript являются значениями, и поэтому нет четкой границы между методами и полями. Если значением свойства является функция, это свойство определяется как метод. В противном случае это обычное свойство, или «поле». Но, несмотря на эти отличия, имеется возможность имитировать все четыре категории членов классов в языке JavaScript. Определение любого класса в языке JavaScript вовлекает три различных объекта (рис. 9.1), а свойства этих трех объектов действуют подобно различным категориям членов класса:

Объект-конструктор

Как уже было отмечено, функция-конструктор (объект) в языке JavaScript определяет имя класса. Свойства, добавляемые в этот объект конструктора, играют роль полей класса и методов класса (в зависимости от того, является ли значение свойства функцией или нет).

Объект-прототип

Свойства этого объекта наследуются всеми экземплярами класса, при этом свойства, значениями которых являются функции, играют роль методов экземпляра класса.

Объект экземпляра

Каждый экземпляр класса – это самостоятельный объект, а свойства, определяемые непосредственно в экземпляре, не являются общими для других экземпляров. Свойства экземпляра, которые не являются функциями, играют роль полей экземпляра класса.

Процесс определения класса в языке JavaScript можно свести к трем этапам. Во-первых, написать функцию-конструктор, которая будет определять свойства экземпляра в новом объекте. Во-вторых, определить методы экземпляров в объекте-прототипе конструктора. В-третьих, определить поля класса и свойства класса в самом конструкторе. Этот алгоритм можно упростить еще больше, определив простую функцию `defineClass()`. (В ней используется функция `extend()` из примера 6.2 с исправлениями из примера 8.3):

```
// Простая функция для определения простых классов
function defineClass(constructor, // Функция, определяющая свойства экземпляра
                     methods,     // Методы экземпляров: копируются в прототип
                     statics)     // Свойства класса: копируются в конструктор
{
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    return constructor;
}

// Простейший вариант нашего класса Range
var SimpleRange =
    defineClass(function(f,t) { this.f = f; this.t = t; },
                {
                    includes: function(x) { return this.f<=x && x <= this.t;},
                    toString: function() { return this.f + "..." + this.t; }
                },
                { upto: function(t) { return new SimpleRange(0, t); } });
```

В примере 9.3 приводится более длинное определение класса. В нем создается класс, представляющий комплексные числа, и демонстрируется, как имитировать члены класса в стиле Java. Здесь все делается «вручную» – без использования функции `defineClass()`, представленной выше.

Пример 9.3. Complex.js: Класс комплексных чисел

```
/*
 * Complex.js:
 * В этом файле определяется класс Complex, представляющий комплексные числа.
 * Напомню, что комплексные числа представляют собой сумму вещественной и мнимой части,
 * где множитель i в мнимой части - это квадратный корень из -1.
 */

/*
 * Функция-конструктор определяет поля экземпляра r и i
 * в каждом создаваемом экземпляре.
 * Эти поля хранят значения вещественной и мнимой частей комплексного числа:
 * они хранят информацию, уникальную для каждого объекта.
 */
function Complex(real, imaginary) {
    if (isNaN(real) || isNaN(imaginary)) // Убедиться, что аргументы - числа.
        throw new TypeError();           // Иначе возбудить исключение.
    this.r = real;                       // Вещественная часть числа.
    this.i = imaginary;                  // Мнимая часть числа.
}

/*
 * Методы экземпляров класса определяются как свойства-функции объекта-прототипа.
 * Методы, определяемые ниже, наследуются всеми экземплярами и обеспечивают общность
 * поведения класса. Обратите внимание, что методы экземпляров в JavaScript
 * должны использовать ключевое слово this для доступа к полям экземпляра.
 */

// Складывает комплексное число that с текущим и возвращает сумму в виде нового объекта.
Complex.prototype.add = function(that) {
    return new Complex(this.r + that.r, this.i + that.i);
};

// Умножает текущее комплексное число на число that и возвращает произведение.
Complex.prototype.mul = function(that) {
    return new Complex(this.r * that.r - this.i * that.i,
                       this.r * that.i + this.i * that.r);
};

// Возвращает вещественный модуль комплексного числа. Он определяется
// как расстояние до числа на комплексной плоскости от точки (0,0).
Complex.prototype.mag = function() {
    return Math.sqrt(this.r*this.r + this.i*this.i);
};

// Возвращает комплексное число с противоположным знаком.
Complex.prototype.neg = function() { return new Complex(-this.r, -this.i); };

// Преобразует объект Complex в строку в понятном формате.
Complex.prototype.toString = function() {
    return "{" + this.r + "," + this.i + "}";
};

// Проверяет равенство данного комплексного числа с заданным.
Complex.prototype.equals = function(that) {
    return that != null &&          // должно быть определено, не равно null
    that.constructor === Complex &&        // и быть экземпляром Complex
    this.r === that.r && this.i === that.i; // и иметь те же значения.
};

/*
 * Поля класса (например, константы) и методы класса определяются как свойства
 * конструктора. Обратите внимание, что в методах класса вообще не используется
 * ключевое слово this: они выполняют операции только со своими аргументами.
 */

// Ниже определяется несколько полей класса, хранящих предопределенные
// комплексные числа. Их имена состоят исключительно из заглавных символов,
// чтобы показать, что они являются константами.
// (В ECMAScript 5 эти свойства можно было бы сделать доступными только для чтения)
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);

// Следующий метод анализирует строку в формате, возвращаемом методом
// экземпляра toString, и возвращает объект Complex или возбуждает исключение TypeError.
Complex.parse = function(s) {
    try {                      // Предполагается, что анализ пройдет успешно
        var m = Complex._format.exec(s); // Регулярное выражение
        return new Complex(parseFloat(m[1]), parseFloat(m[2]));
    } catch (x) {              // Возбудить исключение в случае неудачи
        throw new TypeError("Строка '" + s + "' не может быть преобразована" +
                            " в комплексное число.");
    }
};

// "Частное" поле класса, используемое методом Complex.parse().
// Символ подчеркивания в его имени указывает, что оно предназначено
// для внутреннего использования и не является частью общедоступного API класса.
Complex._format = /^\{([^,]+),([^}]+)\}$/;
```

Определение класса Complex, представленное в примере 9.3, позволяет использовать конструктор, поля экземпляра, методы экземпляров, поля класса и методы класса, как показано ниже:

```
var c = new Complex(2,3);     // Создать новый объект с помощью конструктора
var d = new Complex(c.i,c.r); // Использовать свойства экземпляра с
c.add(d).toString();          // => "{5,5}": использовать методы экземпляров
// Более сложное выражение, в котором используются метод и поле класса
Complex.parse(c.toString()).  // Преобразовать с в строку и обратно,
add(c.neg()).                 // сложить с числом с противоположным знаком,
equals(Complex.ZERO)          // и результат всегда будет равен нулю
```

Несмотря на то что язык JavaScript позволяет имитировать члены классов в стиле языка Java, тем не менее в Java существует множество особенностей, которые не поддерживаются классами в языке JavaScript. Во-первых, в методах экземпляров классов в языке Java допускается использовать поля экземпляра, как если бы они были локальными переменными, – в Java нет необходимости предварять их ссылкой this. В языке JavaScript такая возможность не поддерживается, но похожего эффекта можно добиться с помощью инструкции with (хотя это и не рекомендуется):

```
Complex.prototype.toString = function() {
    with(this) {
        return "{" + r + "," + i + "}";
    }
};
```

В языке Java поддерживается возможность объявлять поля со спецификатором final, чтобы показать, что они являются константами, и объявлять поля и методы со спецификатором private, чтобы показать, что они являются частными для реализации класса и недоступны пользователям класса. В языке JavaScript эти ключевые слова отсутствуют, поэтому, чтобы обозначить частные свойства (имена которых начинаются с символа подчеркивания) и свойства, доступные только для чтения (имена которых содержат только заглавные символы), в примере 9.3 используются соглашения по именованию. Мы еще вернемся к этим двум темам ниже в этой главе: частные свойства можно имитировать с помощью локальных переменных в замыканиях (раздел 9.6.6), а возможность определения свойств-констант поддерживается стандартом ECMAScript 5 (раздел 9.8.2).

9.4. Наращивание возможностей классов

Механизм наследования на основе прототипов, используемый в языке JavaScript, имеет динамическую природу: объекты наследуют все свойства своих прототипов, даже если они были добавлены в прототипы уже после создания объектов. Это означает, что в JavaScript имеется возможность наращивать возможности классов простым добавлением новых методов в объекты-прототипы. Ниже приводится фрагмент, который добавляет метод вычисления сопряженного комплексного числа в класс `Complex` из примера 9.3:

```
// Возвращает комплексное число, которое является сопряженным
// по отношению к текущему.
Complex.prototype.conj = function() { return new Complex(this.r, -this.i); };
```

Объект-прототип встроенных классов JavaScript также «открыт» для подобного наращивания, а это означает, что есть возможность добавлять новые методы к числам, строкам, массивам, функциям и т. д. Данная возможность уже использовалась в примере 8.5. Там мы добавляли метод `bind()` к классу функций в реализации ECMAScript 3, где он отсутствует:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(o /*, аргументы */) {
        // Реализация метода bind...
    };
}
```

Ниже приводятся несколько примеров расширения классов:

```
// Вызывает функцию f в цикле, количество итераций равно самому числу;
// при этом функции каждый раз передается номер итерации
// Например, чтобы вывести "привет" 3 раза:
// var n = 3;
// n.times(function(n) { console.log(n + " привет"); });
Number.prototype.times = function(f, context) {
    var n = Number(this);
    for(var i = 0; i < n; i++) f.call(context, i);
};

// Определяет метод ES5 String.trim(), если он отсутствует.
// Этот метод удаляет пробельные символы в начале и в конце строки и возвращает ее.
String.prototype.trim = String.prototype.trim || function() {
    if (!this) return this;                // Не изменять пустую строку
    return this.replace(/^\s+|\s+$/g, ""); // Регулярное выражение
};

// Возвращает имя функции. Если функция имеет свойство name (нестандартное),
// возвращает его значение. Иначе преобразует функцию в строку и извлекает имя из нее.
// Для неименованных функций возвращает пустую строку.
Function.prototype.getName = function() {
    return this.name || this.toString().match(/function\s*([^(]*)\(/)[1];
};
```

Методы можно также добавлять в `Object.prototype`, тем самым делая их доступными для всех объектов. Однако делать это не рекомендуется, потому что в реализациях, появившихся до ECMAScript 5, отсутствует возможность сделать эти дополнительные методы неперечислимыми. При добавлении новых свойств в `Object.prototype` они становятся доступны для перечисления в любом цикле `for/in`. В разделе 9.8.1 приводится пример использования метода `Object.defineProperty()`, определяемого стандартом ECMAScript 5, для безопасного расширения `Object.prototype`.

Возможность подобного расширения классов, определяемых средой выполнения (такой как веб-броузер), зависит от реализации самой среды. Во многих веб-броузерах, например, допускается добавлять методы в `HTMLElement.prototype`, и такие методы будут наследоваться объектами, представляющими теги HTML в текущем документе. Однако данная возможность не поддерживается в текущей версии Microsoft Internet Explorer, что сильно ограничивает практическую ценность этого приема в клиентских сценариях.

9.5. Классы и типы

В главе 3 уже говорилось, что в языке JavaScript определяется небольшое количество типов: `null`, `undefined`, логические значения, числа, строки, функции и объекты. Оператор `typeof` (раздел 4.13.2) позволяет отличать эти типы. Однако часто бывает желательно интерпретировать каждый класс как отдельный тип данных и иметь возможность отличать объекты разных классов. Отличать встроенные объекты базового языка JavaScript (и объекты среды выполнения в большинстве реализаций клиентского JavaScript) можно по их атрибуту `class` (раздел 6.8.2), используя прием, реализованный в функции `classof()` из примера 6.4. Но когда класс определяется с помощью приемов, продемонстрированных в этой главе, экземпляры объектов всегда содержат в атрибуте `class` значение «Object», поэтому функция `classof()` в данной ситуации оказывается бесполезной.

В следующих подразделах описываются три приема определения класса произвольного объекта: оператор `instanceof`, свойство `constructor` и имя функции-конструктора. Однако ни один из этих приемов не дает полной гарантии, поэтому в разделе 9.5.4 мы обсудим прием грубого определения типа (duck-typing) – философии программирования, в которой центральное место отводится возможностям объекта (т. е. наличию тех или иных методов), а не его принадлежности к какому-либо классу.

9.5.1. Оператор instanceof

Оператор `instanceof` был описан в разделе 4.9.4. Слева от оператора должен находиться объект, для которого выполняется проверка принадлежности к классу, а справа – имя функции-конструктора, представляющей класс. Выражение `o instanceof c` возвращает `true`, если объект `o` наследует `c.prototype`. При этом наследование необязательно может быть непосредственным. Если `o` наследует объект, который наследует объект, наследующий `c.prototype`, выражение все равно вернет `true`.

Как отмечалось выше в этой главе, конструкторы выступают в качестве наружной вывески классов, а фундаментальная идентификация классов проводится прототипами. Несмотря на то что в операторе `instanceof` используется функция-конструктор, этот оператор в действительности проверяет прототип, наследуемый объектом, а не конструктор, с помощью которого он был создан.

Если необходимо проверить, входит ли некоторый определенный прототип в цепочку прототипов объекта без использования функции-конструктора, как промежуточного звена, можно воспользоваться методом `isPrototypeOf()`. Например, ниже показано, как проверить принадлежность объекта `r` к классу `range`, определенному в примере 9.1:

```
range.methods.isPrototypeOf(r); // range.methods - объект-прототип.
```

Один из недостатков оператора `instanceof` и метода `isPrototypeOf()` состоит в том, что они не позволяют узнать класс объекта. Они лишь проверяют принадлежность объекта указанному классу. Еще более серьезные проблемы начинают возникать в клиентских сценариях JavaScript, когда веб-приложение использует несколько окон или фреймов. Каждое окно или фрейм имеет свой собственный контекст выполнения, и в каждом из них имеется свой глобальный объект со своим собственным набором функций-конструкторов. Два массива, созданные в двух разных фреймах, унаследуют идентичные, но разные объекты прототипов, и массив, созданный в одном фрейме, не будет распознаваться оператором `instanceof` как экземпляр конструктора `Array()` в другом фрейме.

9.5.2. Свойство constructor

Другой способ определения класса объекта заключается в использовании свойства `constructor`. Поскольку конструкторы считаются именами классов, определение класса выполняется очень просто. Например:

```
function typeAndValue(x) {
    if (x == null) return ""; // Значения null и undefined не имеют конструкт.
    switch(x.constructor) {
        case Number:  return "Number: " + x;  // Работает с простыми типами
        case String:  return "String: '" + x + "'";
        case Date:    return "Date: " + x;    // Со встроенными типами
        case RegExp:  return "Regexp: " + x;
        case Complex: return "Complex: " + x; // И с пользовательскими типами
    }
}
```

Обратите внимание, что выражения в этом примере, следующие за ключевыми словами `case`, являются функциями. Если бы мы использовали оператор `typeof` или извлекали значение атрибута `class` объекта, они были бы строками.

Для приема, основанного на использовании свойства `constructor`, характерны те же проблемы, что и для приема на основе оператора `instanceof`. Он не всегда будет работать при наличии нескольких контекстов выполнения (например, при наличии нескольких фреймов в окне броузера), совместно использующих общие значения. Каждый фрейм имеет собственный набор функций-конструкторов: конструктор `Array` в одном фрейме не будет считаться идентичным конструктору `Array` в другом фрейме.

Кроме того, язык JavaScript не требует, чтобы каждый объект имел свойство `constructor`: это всего лишь соглашение, по которому по умолчанию объект-прототип создается для каждой функции, и очень просто по ошибке или преднамеренно опустить свойство `constructor` в прототипе. Например, первые два класса в этой главе (в примерах 9.1 и 9.2) были определены так, что их экземпляры не имеют свойства `constructor`.

9.5.3. Имя конструктора

Основная проблема использования оператора `instanceof` или свойства `constructor` для определения класса объекта проявляется при наличии нескольких контекстов выполнения и, соответственно, при наличии нескольких копий функций-конструкторов. Эти функции могут быть совершенно идентичными, но разными объектами, как следствие, не равными друг другу.

Одно из возможных решений проблемы заключается в том, чтобы использовать в качестве идентификатора класса имя функции-конструктора вместо самой функции. Конструктор `Array` в одном окне не будет равен конструктору `Array` в другом окне, но их имена будут равны. Некоторые реализации JavaScript обеспечивают доступ к имени функции через нестандартное свойство `name` объекта функции. Для реализаций, где свойство `name` отсутствует, можно преобразовать функцию в строку и извлечь имя из нее. (Этот прием использовался в разделе 9.4, где демонстрировалось добавление в класс `Function` метода `getName()`.)

В примере 9.4 определяется функция `type()`, возвращающая тип объекта в виде строки. Простые значения и функции она обрабатывает с помощью оператора `typeof`. Для объектов она возвращает либо значение атрибута `class`, либо имя конструктора. В своей работе функция `type()` использует функцию `classof()` из примера 6.4 и метод `Function.getName()` из раздела 9.4. Для простоты реализация этой функции и метода включена в пример.

Пример 9.4. Функция type() для определения типа значения

```
/**
 * Возвращает тип значения в виде строки:
 *   -Если o - null, возвращает "null", если o - NaN, возвращает "nan".
 *   -Если typeof возвращает значение, отличное от "object", возвращает это значение.
 *    (Обратите внимание, что некоторые реализации идентифицируют объекты
 *    регулярных выражений как функции.)
 *   -Если значение атрибута class объекта o отличается от "Object",
 *    возвращает это значение.
 *   -Если o имеет свойство constructor, а конструктор имеет имя, возвращает
 *    имя конструктора.
 *   -Иначе просто возвращает "Object".
 **/
function type(o) {
    var t, c, n; // type, class, name

    // Специальный случай для значения null:
    if (o === null) return "null";

    // Другой специальный случай: NaN - единственное значение, не равное самому себе:
    if (o !== o) return "nan";
    // Применять typeof для любых значений, отличных от "object".
    // Так идентифицируются простые значения и функции.
    if ((t = typeof o) !== "object") return t;

    // Вернуть класс объекта, если это не "Object".
    // Так идентифицируется большинство встроенных объектов.
    if ((c = classof(o)) !== "Object") return c;

    // Вернуть имя конструктора объекта, если имеется
    if (o.constructor && typeof o.constructor === "function" &&
        (n = o.constructor.getName())) return n;

    // Не удалось определить конкретный тип, поэтому остается лишь
    // просто вернуть "Object"
    return "Object";
}

// Возвращает класс объекта.
function classof(o) {
    return Object.prototype.toString.call(o).slice(8,-1);
};

// Возвращает имя функции (может быть "") или null - для объектов,
// не являющихся функциями
Function.prototype.getName = function() {
    if ("name" in this) return this.name;
    return this.name = this.toString().match(/function\s*([^(]*)\(/)[1];
};
```

Этот прием, основанный на использовании имени конструктора для идентификации класса объекта, имеет ту же проблему, что и прием на основе использования свойства `constructor`: не все объекты имеют свойство `constructor`. Кроме того, не все функции имеют имена. Если определить конструктор, используя выражение определения неименованной функции, метод `getName()` будет возвращать пустую строку:

```
// Этот конструктор не имеет имени
var Complex = function(x,y) { this.r = x; this.i = y; }
// Этот конструктор имеет имя
var Range = function Range(f,t) { this.from = f; this.to = t; }
```

9.5.4. Грубое определение типа

Ни один из приемов определения класса объекта, описанных выше, не свободен от проблем, по крайней мере, в клиентском JavaScript. Альтернативный подход состоит в том, чтобы вместо вопроса «какому классу принадлежит объект?» задать вопрос «что может делать этот объект?». Этот подход является типичным в таких языках программирования, как Python и Ruby, и носит название *грубое определение типа* (*duck-typing*, или «утиная типизация») в честь высказывания (часто приписываемого поэту Джеймсу Уиткомбу Райли (James Whitcomb Riley)):

> **Когда я вижу птицу, которая ходит, как утка, плавает, как утка и крякает, как утка, я называю ее уткой.**

Для программистов на языке JavaScript этот афоризм можно интерпретировать так: «Если объект может ходить, плавать и крякать как объект класса Duck, его

можно считать объектом класса Duck, даже если он не наследует объект-прототип класса Duck».

Примером может служить класс `Range` из примера 9.2. Этот класс предназначен для представления диапазонов чисел. Однако обратите внимание, что конструктор `Range()` не проверяет типы аргументов, чтобы убедиться, что они являются числами. Аналогично метод `includes()` использует оператор <=, но не делает никаких предположений о типах значений границ диапазона. Благодаря тому что класс не ограничивается определенным типом значений, его метод `includes()` способен обрабатывать значения границ любых типов, которые могут сравниваться с помощью операторов отношения:

```
var lowercase = new Range("a", "z");
var thisYear = new Range(new Date(2009, 0, 1), new Date(2010, 0, 1));
```

Метод `foreach()` класса `Range` также не проверяет типы значений границ, но он использует функцию `Math.ceil()` и оператор ++, вследствие чего может применяться только к числовым значениям границ диапазона.

В качестве еще одного примера вспомним объекты, подобных массивам, обсуждавшиеся в разделе 7.11. Во многих случаях нам не требуется знать, действительно ли объект является экземпляром класса `Array`: вполне достаточно знать, что он имеет свойство `length` с неотрицательным целочисленным значением. Если посчитать, что целочисленное свойство `length` – это способ массивов «ходить», то мы могли бы сказать, что любой объект, который умеет «ходить» так же, можно (во многих случаях) отнести к массивам.

Однако имейте в виду, что свойство `length` настоящих массивов обладает особым поведением: свойство `length` автоматически обновляется при добавлении нового элемента, а когда значение свойства `length` уменьшается, массив автоматически усекается. Можно было бы сказать, что эти две особенности описывают, как массивы «плавают» и «крякают». Если вы пишете программу, где требуется, чтобы объект «плавал» и «крякал» как массив, вы не сможете использовать в ней объект, который только «ходит» как массив.

Примеры грубого определения типа, представленные выше, опираются на возможность сравнения объектов с помощью оператора < и на особенности поведения свойства `length`. Однако чаще всего под грубым определением типа подразумевается проверка наличия в объекте одного или более методов. Строго типизированная функция `triathlon()` могла бы потребовать, чтобы ее аргумент был объектом класса `TriAthlete`. Альтернативная реализация, выполняющая грубую проверку типа, могла бы принимать любой объект, имеющий методы `walk()`, `swim()` и `bike()`. Если говорить более конкретно, можно было бы переписать класс `Range` так, чтобы вместо операторов < и ++ он использовал бы методы `compareTo()` и `succ()` объектов значений границ.

Можно подойти к определению типа либерально: просто предположить, что входные объекты реализуют все необходимые методы, и не выполнять никаких проверок. Если предположение окажется ошибочным, при попытке вызвать несуществующий метод возникнет ошибка. Другой подход заключается в реализации проверки входных объектов. Однако, вместо того чтобы проверять их принадлежность к определенному классу, можно проверить наличие методов с определенными именами. Это позволит отвергнуть входные объекты раньше и вернуть более информативное сообщение об ошибке.

В примере 9.5 определяется функция quacks() **(более подходящим было бы имя «implements» (реализует), но** implements **является зарезервированным словом), которая может пригодиться для грубого определения типа. Функция** quacks() **проверяет наличие в объекте (первый аргумент функции) методов, указанных в остальных аргументах. Для каждого последующего аргумента, если аргумент является строкой, проверяется наличие метода с этим именем. Если аргумент является объектом, проверяется наличие в первом объекте методов с теми же именами, что и во втором объекте. Если аргумент является функцией, предполагается, что она является конструктором, и в этом случае проверяется наличие в первом объекте методов с теми же именами, что и в объекте-прототипе.**

Пример 9.5. Функция грубой проверки типа

```
// Возвращает true, если o реализует методы, определяемые последующими аргументами.
function quacks(o /*, ... */) {
    for(var i=1; i<arguments.length; i++) {  // для каждого аргумента после o
        var arg = arguments[i];
        switch(typeof arg) {      // Если arg - это:
        case 'string':            // строка: проверить наличие метода с этим именем
            if (typeof o[arg] !== "function") return false;
            continue;
        case 'function':          // функция: использовать объект-прототип
            // Если аргумент является функцией, использовать ее прототип
            arg = arg.prototype;
            // переход к следующему случаю case
        case 'object':            // объект: проверить наличие соотв. методов
            for(var m in arg) {   // Для каждого свойства объекта
                if (typeof arg[m]!=="function") continue; // Пропустить свойства,
                                                          // не являющиеся методами
                if (typeof o[m] !== "function") return false;
            }
        }
    }

    // Если мы попали сюда, значит, объект o реализует все, что требуется
    return true;
}
```

Есть два важных момента, касающиеся функции quacks()**, которые нужно иметь в виду. Во-первых, она просто проверяет наличие в объекте одного или более методов с заданными именами. Присутствие этих свойств ничего не говорит ни о том, что делают эти функции, ни о том, сколько и какого типа аргументы они принимают. Однако это и есть сущность грубого определения типа. Определяя интерфейс, в котором вместо строгой проверки используется прием грубого определения типа, вы получаете более гибкий прикладной интерфейс, но при этом перекладываете на пользователя всю ответственность за правильное его использование. Второй важный момент, касающийся функции** quacks()**, заключается в том, что она не может работать со встроенными классами. Например, нельзя выполнить проверку** quacks(o, Array)**, чтобы убедиться, что объект** o **обладает всеми методами класса** Array**. Это обусловлено тем, что методы встроенных классов недоступны для перечисления и цикл** for/in **в** quacks() **просто не заметит их. (Следует отметить, что это ограничение можно преодолеть в ECMAScript 5 с помощью функции** Object.getOwnProperty Names()**.)**

9.6. Приемы объектно-ориентированного программирования в JavaScript

До сих пор в этой главе мы рассматривали архитектурные основы классов в языке JavaScript: важную роль объектов-прототипов, связь классов с функциями-конструкторами, как действует оператор `instanceof` и т. д. В этом разделе мы продемонстрируем несколько практических (пусть и не фундаментальных) приемов программирования на языке JavaScript с применением классов. Начнем с двух нетривиальных примеров классов, которые не только интересны сами по себе, но также послужат отправной точкой для дальнейшего обсуждения.

9.6.1. Пример: класс множества

Множество – это структура данных, представляющая неупорядоченную коллекцию неповторяющихся значений. К фундаментальным операциям над множествами относятся сложение множеств и проверка вхождения значения в множество, и обычно множества реализуются так, чтобы эти операции имели максимальную скорость выполнения. Объекты в языке JavaScript по сути являются множествами имен свойств, где с каждым именем связано некоторое значение. Таким образом, объекты легко можно использовать как множества строк. В примере 9.6 реализован более универсальный класс `Set`. Он отображает любые значения, допустимые в языке JavaScript, в уникальные строки и использует их в качестве имен свойств. Объекты и функции не имеют достаточно краткого строкового представления, гарантирующего уникальность, поэтому класс `Set` должен определить идентификационное свойство в любом объекте или функции, сохраняемых в множестве.

Пример 9.6. Set.js: произвольное множество значений

```
function Set() {      // Это конструктор
    this.values = {}; // Свойства этого объекта составляют множество
    this.n = 0;       // Количество значений в множестве
    this.add.apply(this, arguments); // Все аргументы являются значениями,
}                                    // добавляемыми в множество

// Добавляет все аргументы в множество.
Set.prototype.add = function() {
    for(var i = 0; i < arguments.length; i++) { // Для каждого аргумента
        var val = arguments[i];                 // Добавляемое значение
        var str = Set._v2s(val);                // Преобразовать в строку
        if (!this.values.hasOwnProperty(str)) { // Если отсутствует в множ.
            this.values[str] = val;             // Отобразить строку в знач.
            this.n++;                           // Увеличить размер множества
        }
    }
    return this;                       // Для поддержки цепочек вызовов методов
};

// Удаляет все аргументы из множества.
Set.prototype.remove = function() {
    for(var i = 0; i < arguments.length; i++) { // Для каждого аргумента
        var str = Set._v2s(arguments[i]);       // Отобразить в строку
        if (this.values.hasOwnProperty(str)) {  // Если присутствует в множ.
            delete this.values[str];            // Удалить
            this.n--;                           // Уменьшить размер множества
        }
    }
    return this;                       // Для поддержки цепочек вызовов методов
};

// Возвращает true, если множество содержит value; иначе возвращает false.
Set.prototype.contains = function(value) {
    return this.values.hasOwnProperty(Set._v2s(value));
};

// Возвращает размер множества.
Set.prototype.size = function() { return this.n; };

// Вызывает функцию f в указанном контексте для каждого элемента множества.
Set.prototype.foreach = function(f, context) {
    for(var s in this.values)                   // Для каждой строки в множестве
        if (this.values.hasOwnProperty(s))      // Пропустить унаследов. свойства
            f.call(context, this.values[s]);    // Вызвать f для значения
};

// Функция для внутреннего использования. Отображает любые значения JavaScript
// в уникальные строки.
Set._v2s = function(val) {
    switch(val) {
        case undefined: return 'u'; // Специальные простые значения
        case null: return 'n';      // отображаются в односимвольные строки.
        case true: return 't';
        case false: return 'f';
        default: switch(typeof val) {
            case 'number': return '#' + val;     // Числа получают префикс #.
            case 'string': return '"' + val;     // Строки получают префикс ".
            default: return '@' + objectId(val); // Объекты и функции - @
        }
    }

    // Для любого объекта возвращается строка. Для разных объектов эта функция
    // будет возвращать разные строки, а для одного и того же объекта всегда
    // будет возвращать одну и ту же строку. Для этого в объекте o создается свойство.
    // В ES5 это свойство можно сделать неперечислимым и доступным только для чтения.
    function objectId(o) {
        var prop = "|**objectid**|";   // Имя частного идентификац. свойства
        if (!o.hasOwnProperty(prop))   // Если объект не имеет этого свойства
            o[prop] = Set._v2s.next++; // Присвоить ему след. доступ. значение
        return o[prop];                // Вернуть идентификатор
    }
};
Set._v2s.next = 100; // Начальное значение для идентификаторов объектов.
```

9.6.2. Пример: типы-перечисления

Перечислениями **называются типы, которые могут принимать конечное количество значений, объявляемых (или «перечисляемых») при определении типа.**

В языке С и его производных типы-перечисления объявляются с помощью ключевого слова `enum`. В ECMAScript 5 `enum` – это зарезервированное (но не используемое) слово, оставленное на тот случай, если когда-нибудь в JavaScript будут реализованы встроенные типы-перечисления. А пока в примере 9.7 демонстрируется, как можно определить собственный тип-перечисление на языке JavaScript. Обратите внимание, что здесь используется функция `inherit()` из примера 6.1.

Пример 9.7 содержит единственную функцию `enumeration()`. Однако она не является конструктором: она не определяет класс с именем «enumeration». Но она является фабричной функцией: при каждом вызове она создает и возвращает новый класс. Ниже показано, как ее можно использовать:

```
// Создать новый класс Coin с четырьмя возможными значениями:
// Coin.Penny, Coin.Nickel и т.д.
var Coin = enumeration({Penny: 1, Nickel:5, Dime:10, Quarter:25});
var c = Coin.Dime;                // Это экземпляр нового класса
c instanceof Coin                 // => true: instanceof работает
c.constructor == Coin             // => true: свойство constructor работает
Coin.Quarter + 3*Coin.Nickel      // => 40: значения преобразуются в числа
Coin.Dime == 10                   // => true: еще одно преобразование в число
Coin.Dime > Coin.Nickel           // => true: операторы отношения работают
String(Coin.Dime) + ":" + Coin.Dime // => "Dime:10": преобразов. в строку
```

Цель этого примера состоит в том, чтобы продемонстрировать, что классы в языке JavaScript являются более гибкими и динамичными, чем статические классы в таких языках, как C++ и Java.

Пример 9.7. Типы-перечисления в JavaScript

```
// Эта функция создает новый тип-перечисление. Объект в аргументе определяет
// имена и значения каждого экземпляра класса. Возвращает функцию-конструктор,
// идентифицирующую новый класс. Отметьте, однако, что конструктор возбуждает
// исключение: его нельзя использовать для создания новых экземпляров типа.
// Возвращаемый конструктор имеет свойства, которые отображают имена в значения,
// а также массив значений values и функцию foreach() для выполнения итераций
function enumeration(namesToValues) {
    // Фиктивный конструктор, который будет использоваться как
    // возвращаемое значение.
    var enumeration = function() { throw "Нельзя создать экземпляр класса" +
                                         " Enumeration"; };

    // Перечислимые значения наследуют объект this.
    var proto = enumeration.prototype = {
        constructor: enumeration,                   // Идентификатор типа
        toString: function() { return this.name; }, // Возвращает имя
        valueOf: function() { return this.value; }, // Возвращает значение
        toJSON: function() { return this.name; }    // Для сериализации
    };

    enumeration.values = []; // Массив перечислимых объектов-значений

    // Теперь создать экземпляры нового типа.
    for(name in namesToValues) {       // Для каждого значения
        var e = inherit(proto);        // Создать объект для его представления
        e.name = name;                 // Дать ему имя
        e.value = namesToValues[name]; // И значение
        enumeration[name] = e;         // Сделать свойством конструктора
        enumeration.values.push(e);    // И сохранить в массиве values
    }
    // Метод класса для обхода экземпляров класса в цикле
    enumeration.foreach = function(f,c) {
        for(var i = 0; i < this.values.length; i++) f.call(c,this.values[i]);
    };

    // Вернуть конструктор, идентифицирующий новый тип
    return enumeration;
}
```

Типичным начальным примером использования типов-перечислений может служить реализация перечисления для представления колоды игральных карт. Пример 9.8 использует функцию `enumeration()` именно для этого, а также определяет классы для представления карт и колод карт.[1]

Пример 9.8. Представление игральных карт в виде типов-перечислений

```
// Определение класса для представления игральной карты
function Card(suit, rank) {
    this.suit = suit;   // Каждая карта имеет масть
    this.rank = rank;   // и значение
}

// Следующие типы-перечисления определяют возможные масти и значения карт
Card.Suit = enumeration({Clubs: 1, Diamonds: 2, Hearts:3, Spades:4});
Card.Rank = enumeration({Two: 2, Three: 3, Four: 4, Five: 5, Six: 6,
                        Seven: 7, Eight: 8, Nine: 9, Ten: 10,
                        Jack: 11, Queen: 12, King: 13, Ace: 14});

// Определение текстового представления карты
Card.prototype.toString = function() {
    return this.rank.toString() + " " + this.suit.toString();
};
// Сравнивает значения двух карт в соответствии с правилами игры в покер
Card.prototype.compareTo = function(that) {
    if (this.rank < that.rank) return -1;
    if (this.rank > that.rank) return 1;
    return 0;
};

// Функция упорядочения карт в соответствии с правилами игры в покер
Card.orderByRank = function(a,b) { return a.compareTo(b); };

// Функция упорядочения карт в соответствии с правилами игры в бридж
Card.orderBySuit = function(a,b) {
    if (a.suit < b.suit) return -1;
    if (a.suit > b.suit) return 1;
    if (a.rank < b.rank) return -1;
    if (a.rank > b.rank) return 1;
```

---

[1] Этот пример основан на примере для языка Java, написанном Джошуа Блохом (Joshua Bloch) и доступном по адресу: *http://jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html*.

```
    return 0;
};

// Определение класса представления стандартной колоды карт
function Deck() {
    var cards = this.cards = [];        // Колода - просто массив карт
    Card.Suit.foreach(function(s) {   // Инициализировать массив
                        Card.Rank.foreach(function(r) {
                                           cards.push(new Card(s,r));
                                       });
                    });
}

// Метод перемешивания: тасует колоду карт и возвращает ее
Deck.prototype.shuffle = function() {
    // Для каждого элемента массива: поменять местами
    // со случайно выбранным элементом ниже
    var deck = this.cards, len = deck.length;
    for(var i = len-1; i > 0; i--) {
        var r = Math.floor(Math.random()*(i+1)), temp;       // Случайное число
        temp = deck[i], deck[i] = deck[r], deck[r] = temp;  // Поменять
    }
    return this;
};

// Метод раздачи: возвращает массив карт
Deck.prototype.deal = function(n) {
    if (this.cards.length < n) throw "Карт для выдачи не хватает";
    return this.cards.splice(this.cards.length-n, n);
};

// Создает новую колоду карт, тасует ее и раздает как в игре в бридж
var deck = (new Deck()).shuffle();
var hand = deck.deal(13).sort(Card.orderBySuit);
```

9.6.3. Стандартные методы преобразований

В разделах 3.8.3 и 6.10 описываются важные методы, используемые для преобразования типа объектов, часть из которых автоматически вызывается интерпретатором JavaScript по мере необходимости. Вам не обязательно определять эти методы в каждом своем классе, но они играют важную роль, и если вы отказываетесь от их реализации в своих классах, это должен быть осознанный выбор, а не простая оплошность.

Первым и наиболее важным является метод `toString()`. Назначение этого метода в том, чтобы возвращать строковое представление объекта. Интерпретатор JavaScript автоматически вызывает этот метод, когда объект используется там, где ожидается строка – в качестве имени свойства, например, или с оператором +, выполняющим конкатенацию строк. Если отказаться от реализации этого метода, ваш класс унаследует от `Object.prototype` реализацию по умолчанию и будет преобразовываться в довольно бесполезную строку «[object Object]». Метод `toString()` может возвращать более удобочитаемую строку, подходящую для отображения на экране перед конечным пользователем вашего приложения. Однако даже если в этом нет необходимости, часто бывает полезно определить свой метод `toString()`, чтобы упростить отладку. Классы `Range` и `Complex`, представленные в примерах 9.2 и 9.3, имеют собственные реализации метода `toString()`, как и типы-перечисления, реализация которых приводится в примере 9.7. Ниже мы определим метод `toString()` для класса `Set` из примера 9.6.

С методом `toString()` тесно связан метод `toLocaleString()`: он должен преобразовывать объект в строку с учетом региональных настроек. По умолчанию объекты наследуют метод `toLocaleString()`, который просто вызывает их метод `toString()`. Некоторые встроенные типы имеют более полезные реализации метода `toLocaleString()`, которые возвращают строки с учетом региональных настроек. Если в реализации своего метода `toString()` вам придется преобразовывать в строки другие объекты, вы также должны определить свой метод `toLocaleString()`, выполняющий те же преобразования вызовом метода `toLocaleString()` объектов. Ниже мы реализуем этот метод для класса `Set`.

Третьим методом является метод `valueOf()`. Его цель – преобразовать объект в простое значение. Метод `valueOf()` вызывается автоматически, когда объект используется в числовом контексте, например, с арифметическими операторами (отличными от +) и с операторами отношения. Большинство объектов не имеют сколько-нибудь осмысленного простого представления и потому не определяют этот метод. Однако типы-перечисления в примере 9.7 представляют случай, когда метод `valueOf()` имеет большое значение.

Четвертый метод – `toJSON()` – вызывается автоматически функцией `JSON.stringify()`. Формат JSON предназначен для сериализации структур данных и может использоваться для представления простых значений, массивов и простых объектов. При преобразовании в этот формат не делается никаких предположений о классах, и при сериализации объекта игнорируются его прототип и конструктор. Если вызвать функцию `JSON.stringify()` для сериализации объекта `Range` или `Complex`, например, она вернет строку вида `{"from":1, "to":3}` или `{"r":1, "i":-1}`. Если передать такую строку функции `JSON.parse()`, она вернет простой объект со свойствами, соответствующими объекту `Range` или `Complex`, но не наследующий методы класса `Range` или `Complex`.

Такой формат сериализации вполне подходит для классов, таких как `Range` и `Complex`, но для более сложных классов может потребоваться написать собственный метод `toJSON()`, чтобы определить иной формат сериализации. Если объект имеет метод `toJSON()`, функция `JSON.stringify()` не будет выполнять сериализацию самого объекта, а вызовет метод `toJSON()` и произведет сериализацию значения (простого значения или объекта), которое он вернет. Например, объекты `Date` имеют собственный метод `toJSON()`, возвращающий строковое представление даты. Типы-перечисления в примере 9.7 делают то же самое: их метод `toJSON()` возвращает то же значение, что и метод `toString()`. Самым близким к представлению множества в формате JSON является массив, поэтому ниже мы определим метод `toJSON()`, который будет преобразовывать объект `Set` в массив значений.

Класс `Set`, представленный в примере 9.6, не определяет ни один из этих методов. Множество не может быть представлено простым значением, поэтому нет смысла определять метод `valueOf()`, но было бы желательно определить в этом классе методы `toString()`, `toLocaleString()` и `toJSON()`. Можно это сделать, как показано ниже. Обратите внимание, что для добавления методов в `Set.prototype` используется функция `extend()` (пример 6.2):

```
// Добавить новые методы в объект-прототип класса Set.
extend(Set.prototype, {
            // Преобразует множество в строку
            toString: function() {
                var s = "{", i = 0;
                this.foreach(function(v) { s += ((i++ > 0)?", ":"") + v; });
                return s + "}";
            },
            // Действует так же, как toString, но вызывает toLocaleString
            // для всех значений
            toLocaleString : function() {
                var s = "{", i = 0;
                this.foreach(function(v) {
                               if (i++ > 0) s += ", ";
                               if (v == null) s += v; // null и undefined
                               else s += v.toLocaleString(); // остальные
                           });
                return s + "}";
            },
            // Преобразует множество в массив значений
            toArray: function() {
                var a = [];
                this.foreach(function(v) { a.push(v); });
                return a;
            }
        });

// Для нужд сериализации в формат JSON интерпретировать множество как массив.
Set.prototype.toJSON = Set.prototype.toArray;
```

9.6.4. Методы сравнения

Операторы сравнения в языке JavaScript сравнивают объекты по ссылке, а не по значению. Так, если имеются две ссылки на объекты, то выясняется, ссылаются они на один и тот же объект или нет, но не выясняется, обладают ли разные объекты одинаковыми свойствами с одинаковыми значениями.[1] Часто бывает удобным иметь возможность сравнить объекты на равенство или определить порядок их следования (например, с помощью операторов отношения < и >). Если вы определяете новый класс и хотите иметь возможность сравнивать экземпляры этого класса, вам придется определить соответствующие методы, выполняющие сравнение.

В языке программирования Java сравнение объектов производится с помощью методов, и подобный подход можно с успехом использовать в JavaScript. Чтобы иметь возможность сравнивать экземпляры класса, можно определить метод экземпляра с именем `equals()`. Этот метод должен принимать единственный аргумент и возвращать `true`, если аргумент эквивалентен объекту, метод которого был вызван. Разумеется, вам решать, что следует понимать под словом «эквивалентен» в контексте вашего класса. Для простых классов часто достаточно просто сравнить свойства `constructor`, чтобы убедиться, что оба объекта имеют один и тот же тип, и затем сравнивать свойства экземпляра двух объектов, чтобы убедиться, что они имеют одинаковые значения. Класс Complex из примера 9.3 как раз обладает таким методом equals(), и для нас не составит труда написать похожий метод для класса Range:

---

[1] То есть являются эквивалентными копиями-экземплярами одного класса. – *Прим. науч. ред.*

```
// Класс Range затирает свое свойство constructor. Поэтому восстановим его.
Range.prototype.constructor = Range;

// Объект Range не может быть равен никакому другому объекту, не являющемуся
// диапазоном значений. Два диапазона равны, только если равны значения их границ.
Range.prototype.equals = function(that) {
    if (that == null) return false; // Отвергнуть null и undefined
    if (that.constructor !== Range) return false; // Отвергнуть  не диапазоны
    // Вернуть true, если значения границ равны.
    return this.from == that.from && this.to == that.to;
}
```

Задание метода equals() для нашего класса Set оказывается несколько сложнее. Мы не можем просто сравнить свойства values двух множеств – требуется выполнить глубокое сравнение:

```
Set.prototype.equals = function(that) {
    // Сокращенная проверка для тривиального случая
    if (this === that) return true;

    // Если объект that не является множеством, он не может быть равен объекту this.
    // Для поддержки подклассов класса Set используется оператор instanceof.
    // Мы могли бы реализовать более либеральную проверку, если бы для нас
    // было желательно использовать прием грубого определения типа.
    // Точно так же можно было бы ужесточить проверку, выполняя сравнение
    //  this.constructor == that.constructor.
    // Обратите внимание, что оператор instanceof корректно отвергает попытки
    // сравнения со значениями null и undefined
    if (!(that instanceof Set)) return false;

    // Если два множества имеют разные размеры, они не равны
    if (this.size() != that.size()) return false;

    // Теперь требуется убедиться, что каждый элемент в this также присутствует в that.
    // Использовать исключение для прерывания цикла foreach, если множества не равны.
    try {
        this.foreach(function(v) { if (!that.contains(v)) throw false; });
        return true;   // Все элементы совпали: множества равны.
    } catch (x) {
        if (x === false) return false; // Элемент в this отсутствует в that.
        throw x;       // Для других исключений: возбудить повторно.
    }
};
```

Иногда бывает полезно реализовать операции сравнения, чтобы выяснить порядок следования объектов. Так, для некоторых классов вполне можно сказать, что один экземпляр «меньше» или «больше» другого. Например, объекты Range можно упорядочивать, опираясь на значение нижней границы. Типы-перечисления можно было бы упорядочивать в алфавитном порядке по именам или в числовом порядке – по значениям, присваиваемым именам (предполагается, что именам присваиваются числовые значения). С другой стороны, объекты класса Set не имеют какого-то естественного порядка следования.

При попытке сравнения объектов с помощью операторов отношения, таких как < и <=, интерпретатор сначала вызовет методы valueOf() объектов и, если методы вернут значения простых типов, сравнит эти значения. Типы-перечисления, возвращаемые методом enumeration() из примера 9.7, имеют метод valueOf() и могут сравниваться с помощью операторов отношения. Однако большинство классов не имеют метода valueOf(). Чтобы сравнивать объекты этих типов для выяснения порядка их следования по вашему выбору, необходимо (опять же, следуя соглашениям, принятым в языке программирования Java) реализовать метод с именем compareTo().

Метод compareTo() должен принимать единственный аргумент и сравнивать его с объектом, метод которого был вызван. Если объект this меньше, чем объект, представленный аргументом, метод compareTo() должен возвращать значение меньше нуля. Если объект this больше, чем объект, представленный аргументом, метод должен возвращать значение больше нуля. И если оба объекта равны, метод должен возвращать ноль. Эти соглашения о возвращаемом значении весьма важны, потому что позволяют выполнять замену операторов отношения следующими выражениями:

Выражение отношения	Выражение замены
`a < b`	`a.compareTo(b) < 0`
`a <= b`	`a.compareTo(b) <= 0`
`a > b`	`a.compareTo(b) > 0`
`a >= b`	`a.compareTo(b) >= 0`
`a == b`	`a.compareTo(b) == 0`
`a != b`	`a.compareTo(b) != 0`

Класс Card в примере 9.8 определяет подобный метод compareTo(), и мы можем написать похожий метод для класса Range, чтобы упорядочивать диапазоны по их нижним границам:

```
Range.prototype.compareTo = function(that) {
    return this.from - that.from;
};
```

Обратите внимание, что вычитание, выполняемое этим методом, возвращает значение меньше нуля, равное нулю или больше нуля в соответствии с порядком следования двух объектов Range. Поскольку перечисление Card.Rank в примере 9.8 имеет метод valueOf(), мы могли бы использовать тот же прием и в методе compareTo() класса Card.

Методы equals(), представленные выше, выполняют проверку типов своих аргументов и возвращают false, как признак неравенства, если аргументы имеют не тот тип. Метод compareTo() не имеет специального возвращаемого значения, с помощью которого можно было бы определить, что «эти два значения не могут сравниваться», поэтому обычно методы compareTo() возбуждают исключение при передаче им аргументов неверного типа.

Примечательно, что метод `compareTo()` класса `Range`, представленный выше, возвращает 0, когда два диапазона имеют одинаковые нижние границы. Это означает, что в данной реализации метод `compareTo()` считает равными любые два диапазона, которые имеют одинаковые нижние границы. Однако такое определение равенства не согласуется с определением, положенным в основу метода `equals()`, который требует равенства обеих границ. Подобные несоответствия в определениях равенства могут стать причиной опасных ошибок, и было бы лучше привести методы `equals()` и `compareTo()` в соответствие друг с другом. Ниже приводится обновленная версия метода `compareTo()` класса `Range`. Он соответствует методу `equals()` и дополнительно возбуждает исключение при передаче ему несопоставимого значения:

```
// Порядок следования диапазонов определяется их нижними границами
// или верхними границами, если нижние границы равны.Возбуждает исключение,
// если методу передается объект, не являющийся экземпляром класса Range.
// Возвращает 0, только если this.equals(that) возвращает true.
Range.prototype.compareTo = function(that) {
    if (!(that instanceof Range))
        throw new Error("Нельзя сравнить Range с " + that);
    var diff = this.from - that.from;         // Сравнить нижние границы
    if (diff == 0) diff = this.to - that.to;  // Если равны, сравнить верхние
    return diff;
};
```

Одна из причин, по которым может потребоваться сравнивать экземпляры класса, – обеспечить возможность сортировки массива экземпляров этого класса. Метод `Array.sort()` может принимать в виде необязательного аргумента функцию сравнения, которая должна следовать тем же соглашениям о возвращаемом значении, что и метод `compareTo()`. При наличии метода `compareTo()`, представленного выше, достаточно просто организовать сортировку массива объектов `Range`, как показано ниже:

```
ranges.sort(function(a,b) { return a.compareTo(b); });
```

Сортировка имеет настолько большое значение, что следует рассмотреть возможность реализации статического метода сравнения в любом классе, где определен метод экземпляров `compareTo()`. Особенно если учесть, что первый может быть легко реализован в терминах второго, например:

```
Range.byLowerBound = function(a,b) { return a.compareTo(b); };
```

При наличии этого метода сортировка массива может быть реализована еще проще:

```
ranges.sort(Range.byLowerBound);
```

Некоторые классы могут быть упорядочены более чем одним способом. Например, класс `Card` определяет один метод класса, упорядочивающий карты по масти, а другой – упорядочивающий по значению.

9.6.5. Заимствование методов

В методах JavaScript нет ничего необычного – это обычные функции, присвоенные свойствам объекта и вызываемые «посредством» или «в контексте» объекта.

Одна и та же функция может быть присвоена двум свойствам и играть роль двух методов. Мы использовали эту возможность в нашем классе Set, например, когда скопировали метод toArray() и заставили его играть вторую роль в виде метода toJSON().

Одна и та же функция может даже использоваться как метод сразу нескольких классов. Большинство методов класса Array, например, являются достаточно универсальными, чтобы копировать их из Array.prototype в прототип своего класса, экземпляры которого являются объектами, подобными массивам. Если вы смотрите на язык JavaScript через призму классических объектно-ориентированных языков, тогда использование методов одного класса в качестве методов другого класса можно рассматривать как разновидность множественного наследования. Однако JavaScript не является классическим объектно-ориентированным языком, поэтому я предпочитаю обозначать такой прием повторного использования методов неофициальным термином *заимствование*.

Заимствоваться могут не только методы класса Array: мы можем реализовать собственные универсальные методы. В примере 9.9 определяются обобщенные методы toString() и equals(), которые с успехом могут использоваться в таких классах, как Range, Complex и Card. Если бы класс Range не имел собственного метода equals(), мы могли бы заимствовать обобщенный метод equals(), как показано ниже:

```
Range.prototype.equals = generic.equals;
```

Обратите внимание, что метод generic.equals() выполняет лишь поверхностное сравнение и не подходит для использования в классах, свойства экземпляров которых ссылаются на объекты с их собственными методами equals(). Отметьте также, что этот метод включает специальный случай для обработки свойства, добавляемого к объектам при включении их в множество Set (пример 9.6).

Пример 9.9. Обобщенные методы, пригодные для заимствования

```
var generic = {
    // Возвращает строку, включающую имя функции-конструктора, если доступно,
    // и имена и значения всех неунаследованных свойств, не являющихся функциями.
    toString: function() {
        var s = '[';
        // Если объект имеет конструктор и конструктор имеет имя, использовать
        // это имя класса как часть возвращаемой строки. Обратите внимание, что
        // свойство name функций является нестандартным и не поддерживается повсеместно.
        if (this.constructor && this.constructor.name)
            s += this.constructor.name + ": ";

        // Теперь обойти все неунаследованные свойства, не являющиеся функциями
        var n = 0;
        for(var name in this) {
            if (!this.hasOwnProperty(name)) continue; // пропустить унаслед.
            var value = this[name];
            if (typeof value === "function") continue; // пропустить методы
            if (n++) s += ", ";
            s += name + '=' + value;
        }
        return s + ']';
    },
    // Проверить равенство, сравнив конструкторы и свойства экземпляров объектов this
    // и that. Работает только с классами, свойства экземпляров которых являются
    // простыми значениями и могут сравниваться с помощью оператора ===.
    // Игнорировать специальное свойство, добавляемое классом Set.
    equals: function(that) {
        if (that == null) return false;
        if (this.constructor !== that.constructor) return false;
        for(var name in this) {
            if (name === "|**objectid**|") continue;     // пропустить спец. св.
            if (!this.hasOwnProperty(name)) continue;   // пропустить унасл. св.
            if (this[name] !== that[name]) return false; // сравнить значения
        }
        return true; // Объекты равны, если все свойства равны.
    }
};
```

9.6.6. Частные члены

В классическом объектно-ориентированном программировании зачастую целью инкапсуляции, или сокрытия данных объектов внутри объектов, является обеспечение доступа к этим данным только через методы объекта и запрет прямого доступа к важным данным. Для достижения этой цели в таких языках, как Java, поддерживается возможность объявления «частных» (private) полей экземпляров класса, доступных только через методы экземпляров класса и невидимые за пределами класса.

Реализовать частные поля экземпляра можно с помощью переменных (или аргументов), хранящихся в замыкании, образуемом вызовом конструктора, который создает экземпляр. Для этого внутри конструктора объявляются функции (благодаря чему она получает доступ к аргументам и локальным переменным конструктора), которые присваиваются свойствам вновь созданного объекта. Этот прием демонстрируется в примере 9.10, где он используется для создания инкапсулированной версии класса `Range`. Вместо простых свойств `from` и `to`, определяющих границы диапазона, экземпляры этой новой версии класса предоставляют методы `from` и `to`, возвращающие значения границ. Методы `from()` и `to()` не наследуются от прототипа, а определяются отдельно для каждого объекта `Range`. Остальные методы класса `Range` определяются в прототипе как обычно, но изменены так, чтобы вместо чтения значений границ напрямую из свойств они вызывали бы методы `from()` и `to()`.

Пример 9.10. Класс Range со слабо инкапсулированными границами

```
function Range(from, to) {
    // Не сохраняет границы в свойствах объекта. Вместо этого определяет функции доступа,
    // возвращающие значения границ. Сами значения хранятся в замыкании.
    this.from = function() { return from; };
    this.to = function() { return to; };
}

// Методы прототипа не имеют прямого доступа к границам: они должны вызывать
// методы доступа, как любые другие функции и методы.
Range.prototype = {
    constructor: Range,
    includes: function(x) { return this.from() <= x && x <= this.to(); },
    foreach: function(f) {
        for(var x=Math.ceil(this.from()), max=this.to(); x <= max; x++) f(x);
    },
    toString: function() { return "(" + this.from() + "..." + this.to() + ")"; }
};
```

Новый класс `Range` определяет методы для чтения значений границ диапазона, но в нем отсутствуют методы или свойства для изменения этих значений. Это обстоятельство делает экземпляры этого класса *неизменяемыми*: при правильном использовании границы объекта `Range` не должны изменяться после его создания. Однако если не использовать возможности ECMAScript 5 (раздел 9.8.3), свойства `from` и `to` по-прежнему остаются доступными для записи и в действительности объекты `Range` не являются неизменяемыми:

```
var r = new Range(1,5);              // "неизменяемый" диапазон
r.from = function() { return 0; };  // Изменчивость имитируется заменой метода
```

Имейте в виду, что такой прием инкапсуляции имеет отрицательные стороны. Класс, использующий замыкание для инкапсуляции, практически наверняка будет работать медленнее и занимать больше памяти, чем эквивалент с простыми свойствами.

9.6.7. Перегрузка конструкторов и фабричные методы

Иногда бывает необходимо реализовать возможность инициализации объектов несколькими способами. Например, можно было бы предусмотреть возможность инициализации объекта `Complex` значениями радиуса и угла (полярные координаты) вместо вещественной и мнимой составляющих. Или создавать объекты множеств `Set`, членами которых являются элементы массива, а не аргументы конструктора.

Один из способов реализовать такую возможность заключается в создании *перегруженных* версий конструктора и выполнении в них различных способов инициализации в зависимости от передаваемых аргументов. Ниже приводится пример перегруженной версии конструктора `Set`:

```
function Set() {
    this.values = {}; // Свойство для хранения множества
    this.n = 0;       // Количество значений в множестве

    // Если конструктору передается единственный объект, подобный массиву,
    // он добавляет элементы массива в множество.
    // В противном случае в множество добавляются все аргументы
    if (arguments.length == 1 && isArrayLike(arguments[0]))
        this.add.apply(this, arguments[0]);
    else if (arguments.length > 0)
        this.add.apply(this, arguments);
}
```

Такое определение конструктора `Set()` позволяет явно перечислять элементы множества в вызове конструктора или передавать ему массив элементов множества. К сожалению, этот конструктор имеет одну неоднозначность: его нельзя

использовать для создания множества, содержащего единственный элемент – массив. (Для этого потребуется создать пустое множество, а затем явно вызвать метод `add()`.)

В случае с комплексными числами реализовать перегруженную версию конструктора, которая инициализирует объект полярными координатами, вообще невозможно. Оба представления комплексных чисел состоят из двух вещественных чисел. Если не добавить в конструктор третий аргумент, он не сможет по своим аргументам определить, какое представление следует использовать. Однако вместо этого можно написать фабричный метод – метод класса, возвращающий экземпляр класса. Ниже приводится фабричный метод, возвращающий объект `Complex`, инициализированный полярными координатами:

```
Complex.polar = function(r, theta) {
    return new Complex(r*Math.cos(theta), r*Math.sin(theta));
};
```

А так можно реализовать фабричный метод для инициализации объекта `Set` массивом:

```
Set.fromArray = function(a) {
    s = new Set();      // Создать пустое множество
    s.add.apply(s, a); // Передать элементы массива методу add
    return s;           // Вернуть новое множество
};
```

Привлекательность фабричных методов заключается в том, что им можно дать любые имена, а методы с разными именами могут выполнять разные виды инициализации. Поскольку конструкторы играют роль имен классов, обычно в классах определяется единственный конструктор. Однако это не является непреложным правилом. В языке JavaScript допускается определять несколько функций-конструкторов, использующих общий объект-прототип, и в этом случае объекты, созданные разными конструкторами одного класса, будут иметь один и тот же тип. Данный прием не рекомендуется к использованию, тем не менее ниже приводится вспомогательный конструктор такого рода:

```
// Вспомогательный конструктор для класса Set.
function SetFromArray(a) {
    // Инициализировать новый объект вызовом конструктора Set() как функции,
    // передав ей элементы массива в виде отдельных аргументов.
    Set.apply(this, a);
}
// Установить прототип, чтобы функция SetFromArray создавала экземпляры Set
SetFromArray.prototype = Set.prototype;

var s = new SetFromArray([1,2,3]);
s instanceof Set // => true
```

В ECMAScript 5 функции имеют метод `bind()`, особенности которого позволяют создавать подобные вспомогательные конструкторы (раздел 8.7.4).

9.7. Подклассы

В объектно-ориентированном программировании класс `B` может *расширять*, или *наследовать*, другой класс `A`. Класс `A` в этом случае называется *суперклассом*, а класс `B` – *подклассом*. Экземпляры класса `B` наследуют все методы экземпляров класса `A`. Класс `B` может определять собственные методы экземпляров, некоторые из которых могут *переопределять* методы класса `A` с теми же именами. Если метод класса `B` переопределяет метод класса `A`, переопределяющий метод класса `B` может вызывать переопределенный метод класса `A`: этот прием называется *вызовом метода базового класса*. Аналогично конструктор `B()` подкласса может вызывать конструктор `A()` суперкласса. Это называется *вызовом конструктора базового класса*. Подклассы сами могут наследоваться другими подклассами и, работая над созданием иерархии классов, иногда бывает полезно определять *абстрактные классы*. Абстрактный класс – это класс, в котором объявляется один или более методов без реализации. Реализация таких *абстрактных методов* возлагается на *конкретные подклассы* абстрактного класса.

Ключом к созданию подклассов в языке JavaScript является корректная инициализация объекта-прототипа. Если класс `B` расширяет класс `A`, то объект `B.prototype` должен наследовать `A.prototype`. В этом случае экземпляры класса `B` будут наследовать свойства от объекта `B.prototype`, который в свою очередь наследует свойства от `A.prototype`. В этом разделе демонстрируются все представленные выше термины, связанные с подклассами, а также рассматривается прием, альтернативный наследованию, который называется *композицией*.

Используя в качестве основы класс `Set` из примера 9.6, этот раздел продемонстрирует, как определять подклассы, как вызывать конструкторы базовых классов и переопределенные методы, как вместо наследования использовать прием композиции и, наконец, как отделять интерфейсы от реализации с помощью абстрактных классов. Этот раздел завершает расширенный пример, в котором определяется иерархия классов `Set`. Следует отметить, что первые примеры в этом разделе служат цели продемонстрировать основные приемы использования механизма наследования, и некоторые из них имеют существенные недостатки, которые будут обсуждаться далее в этом же разделе.

9.7.1. Определение подкласса

В языке JavaScript объекты наследуют свойства (обычно методы) от объекта-прототипа своего класса. Если объект `O` является экземпляром класса `B`, а класс `B` является подклассом класса `A`, то объект `O` также наследует свойства класса `A`. Добиться этого можно за счет наследования объектом-прототипом класса `B` свойств объекта-прототипа класса `A`, как показано ниже, с использованием функции `inherit()` (пример 6.1):

```
B.prototype = inherit(A.prototype); // Подкласс наследует суперкласс
B.prototype.constructor = B;        // Переопределить унаследованное св. constructor
```

Эти две строки являются ключом к созданию подклассов в JavaScript. Без них объект-прототип будет обычным объектом – объектом, наследующим свойства от `Object.prototype`, – а это означает, что класс будет подклассом класса `Object`, подобно всем остальным классам. Если добавить эти две строки в функцию `defineClass()`

(раздел 9.3), ее можно будет преобразовать в функцию `defineSubclass()` **и в метод** `Function.prototype.extend()`**, как показано в примере 9.11.**

Пример 9.11. Вспомогательные инструменты определения подклассов

```
// Простая функция для создания простых подклассов
function defineSubclass(superclass,  // Конструктор суперкласса
                        constructor, // Конструктор нового подкласса
                        methods,     // Методы экземпл.: копируются в прототип
                        statics)     // Свойства класса: копируются в констр-р
{
    // Установить объект-прототип подкласса
    constructor.prototype = inherit(superclass.prototype);
    constructor.prototype.constructor = constructor;
    // Скопировать методы methods и statics, как в случае с обычными классами
    if (methods) extend(constructor.prototype, methods);
    if (statics) extend(constructor, statics);
    // Вернуть класс
    return constructor;
}

// То же самое можно реализовать в виде метода конструктора суперкласса
Function.prototype.extend = function(constructor, methods, statics) {
    return defineSubclass(this, constructor, methods, statics);
};
```

Пример 9.12 демонстрирует, как определить подкласс «вручную», без использования функции `defineSubclass()`**. В этом примере определяется подкласс** `SingletonSet` **класса** `Set`**. Класс** `SingletonSet` **представляет специализированное множество, доступное только для чтения и состоящее из единственного постоянно элемента.**

Пример 9.12. SingletonSet: простой подкласс множеств

```
// Функция-конструктор
function SingletonSet(member) {
    this.member = member; // Сохранить единственный элемент множества
}

// Создает объект-прототип, наследующий объект-прототип класса Set.
SingletonSet.prototype = inherit(Set.prototype);

// Далее добавляются свойства в прототип.
// Эти свойства переопределяют одноименные свойства объекта Set.prototype.
extend(SingletonSet.prototype, {
    // Установить свойство constructor
    constructor: SingletonSet,
    // Данное множество доступно только для чтения: методы add() и remove()
    // возбуждают исключение
    add: function() { throw "множество доступно только для чтения"; },
    remove: function() { throw "множество доступно только для чтения"; },
    // Экземпляры SingletonSet всегда имеют размер, равный 1
    size: function() { return 1; },
    // Достаточно вызвать функцию один раз и передать ей единственный элемент.
    foreach: function(f, context) { f.call(context, this.member); },
    // Метод contains() стал проще: такая реализация пригодна только
    // для множества с единственным элементом
    contains: function(x) { return x === this.member; }
});
```

Класс `SingletonSet` имеет очень простую реализацию, состоящую из пяти простых методов. Этот класс не только реализует пять основных методов класса `Set`, но и наследует от своего суперкласса такие методы, как `toString()`, `toArray()` и `equals()`. Возможность наследования методов является одной из основных причин определения подклассов. Метод `equals()` класса `Set` (определен в разделе 9.6.4), например, может сравнивать любые экземпляры класса `Set`, имеющие методы `size()` и `foreach()`, с любыми экземплярами класса `Set`, имеющими методы `size()` и `contains()`. Поскольку класс `SingletonSet` является подклассом класса `Set`, он автоматически наследует его метод `equals()` и не обязан иметь собственную реализацию этого метода. Безусловно, учитывая чрезвычайно упрощенную структуру множества, содержащего единственный элемент, можно было бы реализовать для класса `SingletonSet` более эффективную версию метода `equals()`:

```
SingletonSet.prototype.equals = function(that) {
    return that instanceof Set && that.size()==1 && that.contains(this.member);
};
```

Обратите внимание, что класс `SingletonSet` не просто заимствует список методов из класса `Set`: он динамически наследует методы класса `Set`. Если в `Set.prototype` добавить новый метод, он тут же станет доступен всем экземплярам классов `Set` и `SingletonSet` (в предположении, что класс `SingletonSet` не определяет собственный метод с таким же именем).

9.7.2. Вызов конструктора и методов базового класса

Класс `SingletonSet` из предыдущего раздела определяет совершенно новый тип множеств и полностью переопределяет основные методы, наследуемые от суперкласса. Однако часто при определении подкласса необходимо лишь расширить или немного изменить поведение методов суперкласса, а не заменить их полностью. В этом случае конструктор и методы подкласса могут вызывать конструктор и методы базового класса.

Пример 9.13 демонстрирует применение этого приема. Он определяет подкласс `NonNullSet` класса `Set`: тип множеств, которые не могут содержать элементы со значениями `null` и `undefined`. Чтобы исключить возможность включения в множество таких элементов, класс `NonNullSet` должен выполнить в методе `add()` проверку значений добавляемых элементов на равенство значениям `null` и `undefined`. Но при этом не требуется включать в класс полную реализацию метода `add()` – можно просто вызвать версию метода из суперкласса. Обратите также внимание, что конструктор `NonNullSet()` тоже не реализует все необходимые операции: он просто передает свои аргументы конструктору суперкласса (вызывая его как функцию, а не как конструктор), чтобы конструктор суперкласса мог инициализировать вновь созданный объект.

Пример 9.13. Вызов из подкласса конструктора и метода базового суперкласса

```
/*
 * NonNullSet - подкласс класса Set, который не может содержать элементы
 * со значениями null и undefined.
 */
```

```
function NonNullSet() {
    // Простое обращение к суперклассу.
    // Вызвать конструктор суперкласса как обычную функцию для инициализации
    // объекта, который был создан вызовом этого конструктора.
    Set.apply(this, arguments);
}

// Сделать класс NonNullSet подклассом класса Set:
NonNullSet.prototype = inherit(Set.prototype);
NonNullSet.prototype.constructor = NonNullSet;

// Чтобы исключить возможность добавления значений null и undefined,
// достаточно переопределить метод add()
NonNullSet.prototype.add = function() {
    // Проверить наличие аргументов со значениями null и undefined
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] == null)
            throw new Error("Нельзя добавить null или undefined в NonNullSet");

    // Вызвать метод базового суперкласса, чтобы фактически добавить элементы
    return Set.prototype.add.apply(this, arguments);
};
```

Теперь обобщим понятие «множество без пустых элементов» до понятия «фильтрованное множество»: множество, элементы которого должны пропускаться через функцию-фильтр перед добавлением. Определим фабричную функцию (подобную функции `enumeration()` из примера 9.7), которая будет получать функцию-фильтр и возвращать новый подкласс класса `Set`. В действительности можно пойти еще дальше по пути обобщений и определить фабричную функцию, принимающую два аргумента: наследуемый класс и функцию-фильтр, применяемую к методу `add()`. Новой фабричной функции можно было бы дать имя `filteredSetSubclass()` и использовать ее, как показано ниже:

```
// Определить класс множеств, которые могут хранить только строки
var StringSet = filteredSetSubclass(Set,
                                    function(x) {return typeof x==="string";});

// Определить класс множеств, которые не могут содержать значения null,
// undefined и функции
var MySet = filteredSetSubclass(NonNullSet,
                                function(x) {return typeof x !== "function";});
```

Реализация этой фабричной функции приводится в примере 9.14. Обратите внимание, что эта функция вызывает метод и конструктор базового класса подобно тому, как это реализовано в классе `NonNullSet`.

Пример 9.14. Вызов конструктора и метода базового класса

```
/*
 * Эта функция возвращает подкласс указанного класса Set и переопределяет
 * метод add() этого класса, применяя указанный фильтр.
 */
function filteredSetSubclass(superclass, filter) {
    var constructor = function() {          // Конструктор подкласса
        superclass.apply(this, arguments); // Вызов конструктора базового класса
    };
    var proto = constructor.prototype = inherit(superclass.prototype);
    proto.constructor = constructor;
    proto.add = function() {
        // Примерить фильтр ко всем аргументам перед добавлением
        for(var i = 0; i < arguments.length; i++) {
            var v = arguments[i];
            if (!filter(v)) throw("значение " + v + " отвергнуто фильтром");
        }
        // Вызвать реализацию метода add из базового класса
        superclass.prototype.add.apply(this, arguments);
    };
    return constructor;
}
```

В примере 9.14 есть один интересный момент, который хотелось бы отметить. Он заключается в том, что, обертывая операцию создания подкласса функцией, мы получаем возможность использовать аргумент `superclass` в вызовах конструктора и метода базового класса и избежать указания фактического имени суперкласса. Это означает, что в случае изменения имени суперкласса достаточно будет изменить имя в одном месте, а не отыскивать все его упоминания в программном коде. Такого способа стоит придерживаться даже в случаях, не связанных с определением фабричных функций. Например, с помощью функции-обертки можно было бы переписать определение класса `NonNullSet` и метода `Function.prototype.extend()` (пример 9.11), как показано ниже:

```
var NonNullSet = (function() { // Определить и вызвать функцию
    var superclass = Set;      // Имя суперкласса указывается в одном месте.
    return superclass.extend(
        function() { superclass.apply(this, arguments); }, // конструктор
        {                                                  // методы
            add: function() {
                // Проверить аргументы на равенство null или undefined
                for(var i = 0; i < arguments.length; i++)
                    if (arguments[i] == null)
                        throw new Error("Нельзя добавить null или undefined");

                // Вызвать метод базового класса, чтобы выполнить добавление
                return superclass.prototype.add.apply(this, arguments);
            }
        });
}());
```

В заключение хотелось бы подчеркнуть, что возможность создания подобных фабрик классов обусловлена динамической природой языка JavaScript. Фабрики классов представляют собой мощный и гибкий инструмент, не имеющий аналогов в языках, подобных Java и C++.

9.7.3. Композиция в сравнении с наследованием

В предыдущем разделе мы решили определить класс множеств, накладывающий ограничения на свои элементы в соответствии с некоторыми критериями, а для достижения поставленной цели использовали механизм наследования, определив функцию создания специализированного подкласса указанной реализации множества, использующего указанную функцию-фильтр для ограничения круга допустимых элементов множества. При таком подходе для каждой комбинации суперкласса и функции-фильтра необходимо создавать новый класс.

Однако существует более простой путь решения этой задачи. В объектно-ориентированном программировании существует известный принцип «предпочтения композиции перед наследованием».[1] В данном случае применение приема композиции могло бы выглядеть как реализация нового множества, «обертывающего» другой объект множества и делегирующего ему все обращения после фильтрации нежелательных элементов. Пример 9.15 демонстрирует, как это реализовать.

Пример 9.15. Композиция множеств вместо наследования

```
/*
 * Объект FilteredSet обертывает указанный объект множества и применяет
 * указанный фильтр в своем методе add(). Обращения ко всем остальным базовым
 * методам просто передаются обернутому экземпляру множества.
 */
var FilteredSet = Set.extend(
    function FilteredSet(set, filter) { // Конструктор
        this.set = set;
        this.filter = filter;
    },
    {                                    // Методы экземпляров
        add: function() {
            // Если фильтр был указан, применить его
            if (this.filter) {
                for(var i = 0; i < arguments.length; i++) {
                    var v = arguments[i];
                    if (!this.filter(v))
                        throw new Error("FilteredSet: значение " + v +
                                        " отвергнуто фильтром");
                }
            }

            // Затем вызвать метод add() объекта this.set.add()
            this.set.add.apply(this.set, arguments);
            return this;
        },
        // Остальные методы просто вызывают соответствующие
        // методы объекта this.set и ничего более.
        remove: function() {
            this.set.remove.apply(this.set, arguments);
            return this;
        },
        contains: function(v) { return this.set.contains(v); },
        size: function() { return this.set.size(); },
        foreach: function(f,c) { this.set.foreach(f,c); }
    });
```

---

[1] См. Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования» или Дж. Блох «Java. Эффективное программирование».

Одно из преимуществ применения приема композиции в данном случае заключается в том, что требуется определить только один подкласс `FilteredSet`. Экземпляры этого класса могут накладывать ограничения на элементы любого другого экземпляра множества. Например, вместо класса `NonNullSet`, представленного выше, реализовать подобные ограничения можно было бы так:

```
var s = new FilteredSet(new Set(), function(x) { return x !== null; });
```

Можно даже наложить еще один фильтр на фильтрованное множество:

```
var t = new FilteredSet(s, { function(x} { return !(x instanceof Set); });
```

9.7.4. Иерархии классов и абстрактные классы

В предыдущем разделе было предложено «предпочесть композицию наследованию». Но для иллюстрации этого принципа мы создали подкласс класса `Set`. Сделано это было для того, чтобы получившийся подкласс был `instanceof Set` и наследовал полезные методы класса `Set`, такие как `toString()` и `equals()`. Это достаточно уважительные причины, но, тем не менее, было бы неплохо иметь возможность использовать прием композиции без необходимости наследовать некоторую определенную реализацию множества, такую как класс `Set`. Аналогичный подход можно было бы использовать и при создании класса `SingletonSet` (пример 9.12) – этот класс был определен как подкласс класса `Set`, чтобы унаследовать вспомогательные методы, но его реализация существенно отличается от реализации суперкласса. Класс `SingletonSet` – это не специализированная версия класса `Set`, а совершенно иной тип множеств. В иерархии классов `SingletonSet` должен был бы находиться на одном уровне с классом `Set`, а не быть его потомком.

Решение этой проблемы в классических объектно-ориентированных языках, а также в языке JavaScript заключается в том, чтобы отделить интерфейс от реализации. Представьте, что мы определили класс `AbstractSet`, реализующий вспомогательные методы, такие как `toString()`, в котором отсутствуют реализации базовых методов, таких как `foreach()`. Тогда все наши реализации множеств – `Set`, `SingletonSet` и `FilteredSet` – могли бы наследовать класс `AbstractSet`. При этом классы `FilteredSet` и `SingletonSet` больше не наследовали бы ненужные им реализации.

Пример 9.16 развивает этот подход еще дальше и определяет иерархию абстрактных классов множеств. Класс `AbstractSet` определяет только один абстрактный метод, `contains()`. Любой класс, который претендует на роль множества, должен будет определить хотя бы один этот метод. Далее в примере определяется класс `AbstractEnumerableSet`, наследующий класс `AbstractSet`. Этот класс определяет абстрактные методы `size()` and `foreach()` и реализует конкретные вспомогательные методы (`toString()`, `toArray()`, `equals()` и т. д.). `AbstractEnumerableSet` не определяет методы `add()` или `remove()` и представляет класс множеств, доступных только для чтения. Класс `SingletonSet` может быть реализован как конкретный подкласс. Наконец, в примере определяется класс `AbstractWritableSet`, наследующий `AbstractEnumerableSet`. Этот последний абстрактный класс определяет абстрактные методы `add()` и `remove()` и реализует конкретные методы, такие как `union()` и `intersection()`, использующие их. Класс `AbstractWritableSet` отлично подходит на роль суперкласса для наших классов `Set` и `FilteredSet`. Однако они не были добавлены в пример, а вместо них была включена новая конкретная реализация с именем `ArraySet`.

Пример 9.16 довольно объемен, но он заслуживает детального изучения. Обратите внимание, что для простоты создания подклассов в нем используется функция `Function.prototype.extend()`**.**

Пример 9.16. Иерархия абстрактных и конкретных классов множеств

```
// Вспомогательная функция, которая может использоваться для определения
// любого абстрактного метода
function abstractmethod() { throw new Error("абстрактный метод"); }

/*
 * Класс AbstractSet определяет единственный абстрактный метод, contains().
 */
function AbstractSet() {
    throw new Error("Нельзя создать экземпляр абстрактного класса");
}
AbstractSet.prototype.contains = abstractmethod;

/*
 * NotSet - конкретный подкласс класса AbstractSet.
 * Элементами этого множества являются все значения, которые не являются
 * элементами некоторого другого множества. Поскольку это множество
 * определяется в терминах другого множества, оно не доступно для записи,
 * а так как оно имеет бесконечное число элементов, оно недоступно для перечисления.
 * Все, что позволяет этот класс, - это проверить принадлежность к множеству.
 * Обратите внимание, что для определения этого подкласса используется метод
 * Function.prototype.extend(), объявленный выше.
 */
var NotSet = AbstractSet.extend(
    function NotSet(set) { this.set = set; },
    {
        contains: function(x) { return !this.set.contains(x); },
        toString: function(x) { return "~" + this.set.toString(); },
        equals: function(that) {
            return that instanceof NotSet && this.set.equals(that.set);
        }
    }
);

/*
 * AbstractEnumerableSet - абстрактный подкласс класса AbstractSet.
 * Определяет абстрактные методы size() и foreach() и реализует конкретные
 * методы isEmpty(), toArray(), to[Locale]String() и equals().
 * Подклассы, реализующие методы contains(), size() и foreach(),
 * получают эти пять конкретных методов даром.
 */
var AbstractEnumerableSet = AbstractSet.extend(
    function() {
        throw new Error("Нельзя создать экземпляр абстрактного класса");
    },
    {
        size: abstractmethod,
        foreach: abstractmethod,
        isEmpty: function() { return this.size() == 0; },
        toString: function() {
            var s = "{", i = 0;
            this.foreach(function(v) {
                             if (i++ > 0) s += ", ";
                             s += v;
                      });
            return s + "}";
        },
        toLocaleString : function() {
            var s = "{", i = 0;
            this.foreach(function(v) {
                             if (i++ > 0) s += ", ";
                             if (v == null) s += v; // null и undefined
                             else s += v.toLocaleString(); // все остальные
                      });
            return s + "}";
        },
        toArray: function() {
            var a = [];
            this.foreach(function(v) { a.push(v); });
            return a;
        },
        equals: function(that) {
            if (!(that instanceof AbstractEnumerableSet)) return false;
            // Если множество that имеет другой размер, множества не равны
            if (this.size() != that.size()) return false;
            // Проверить наличие каждого элемента this в множестве that.
            try {
                this.foreach(function(v){
                             if (!that.contains(v)) throw false;}
                           );
                return true;   // Все элементы одинаковые: множества равны.
            } catch (x) {
                if (x === false) return false; // Множества не равны
                    throw x;   // Повторно возбудить любое иное возникшее исключение.
            }
        }
    });

/*
 * SingletonSet - конкретный подкласс класса AbstractEnumerableSet.
 * Множество из единственного элемента, доступное только для чтения.
 */
var SingletonSet = AbstractEnumerableSet.extend(
    function SingletonSet(member) { this.member = member; },
    {
        contains: function(x) { return x === this.member; },
        size: function() { return 1; },
        foreach: function(f,ctx) { f.call(ctx, this.member); }
    }
);

/*
 * AbstractWritableSet - абстрактный подкласс класса AbstractEnumerableSet.
 * Определяет абстрактные методы add() и remove() и реализует конкретные
 * методы union(), intersection() и difference().
 */
var AbstractWritableSet = AbstractEnumerableSet.extend(
    function() {
        throw new Error("Нельзя создать экземпляр абстрактного класса");
    },
    {
        add: abstractmethod,
        remove: abstractmethod,
        union: function(that) {
            var self = this;
            that.foreach(function(v) { self.add(v); });
            return this;
        },
        intersection: function(that) {
            var self = this;
            this.foreach(function(v){ if(!that.contains(v)) self.remove(v);});
            return this;
        },
        difference: function(that) {
            var self = this;
            that.foreach(function(v) { self.remove(v); });
            return this;
        }
    });

/*
 * ArraySet - конкретный подкласс класса AbstractWritableSet.
 * Представляет множество элементов как массив значений и реализует линейный
 * поиск в массиве в своем методе contains(). Поскольку алгоритм метода contains()
 * имеет сложность O(n) вместо O(1), данный класс следует использовать только
 * для создания относительно небольших множеств.
 * Обратите внимание, что эта реализация опирается на методы класса Array
 * indexOf() и forEach(), которые определяются стандартом ES5.
 */
var ArraySet = AbstractWritableSet.extend(
    function ArraySet() {
        this.values = [];
        this.add.apply(this, arguments);
    },
    {
        contains: function(v) { return this.values.indexOf(v) != -1; },
        size: function() { return this.values.length; },
        foreach: function(f,c) { this.values.forEach(f, c); },
        add: function() {
            for(var i = 0; i < arguments.length; i++) {
                var arg = arguments[i];
                if (!this.contains(arg)) this.values.push(arg);
            }
            return this;
        },
        remove: function() {
            for(var i = 0; i < arguments.length; i++) {
                var p = this.values.indexOf(arguments[i]);
                if (p == -1) continue;
                this.values.splice(p, 1);
            }
            return this;
        }
    }
);
```

9.8. Классы в ECMAScript 5

Стандарт ECMAScript 5 добавляет методы, позволяющие определять атрибуты свойств (методы чтения и записи, а также признаки доступности для перечисления, записи и настройки) и ограничивать возможность расширения объектов. Эти методы были описаны в разделах 6.6, 6.7 и 6.8.3 и могут пригодиться при определении классов. В следующих подразделах демонстрируется, как использовать новые возможности ECMAScript 5 для повышения надежности своих классов.

9.8.1. Определение неперечислимых свойств

Класс `Set`, представленный в примере 9.6, вынужден использовать уловку, чтобы обеспечить возможность сохранения объектов: он добавляет свойство «object id» ко всем объектам, добавляемым в множество. Если позднее в каком-то другом месте программы будет выполнен обход свойств этого объекта с помощью цикла `for/in`, это свойство будет обнаружено. Стандарт ECMAScript 5 позволяет исключить такую возможность, сделав свойство неперечислимым. В примере 9.17 демонстрируется, как это сделать с помощью `Object.defineProperty()`, а также показывает, как определить метод чтения и как проверить возможность расширения объекта.

Пример 9.17. Определение неперечислимых свойств

```
// Обертывание программного код функцией позволяет определять переменные
// в области видимости функции
(function() {
    // Определить свойство objectId как неперечислимое и наследуемое
    // всеми объектами. При попытке получить значение этого свойства
    // вызывается метод чтения. Свойство не имеет метода записи, поэтому
    // оно доступно только для чтения. Свойство определяется как ненастраиваемое,
    // поэтому его нельзя удалить.
    Object.defineProperty(Object.prototype, "objectId", {
                           get: idGetter,       // Метод чтения значения
                           enumerable: false,   // Неперечислимое
                           configurable: false  // Не может быть удалено
                     });

    // Функция чтения, которая вызывается при попытке получить значение
    // свойства objectId
    function idGetter() {         // Функция чтения, возвращающая id
        if (!(idprop in this)) { // Если объект еще не имеет id
            if (!Object.isExtensible(this)) // И если можно добавить свойство
                throw Error("Нельзя определить id нерасширяемого объекта");
            Object.defineProperty(this, idprop, {      // Добавить его.
                                  value: nextid++,     // Значение
```

```
                                   writable: false,    // Только для чтения
                                   enumerable: false,  // Неперечислимое
                                   configurable: false // Неудаляемое
                               });
        }
        return this[idprop]; // Вернуть существующее или новое значение
    };

    // Следующие переменные используются функцией idGetter() и являются
    // частными для этой функции
        var idprop = "|**objectId**|";  // Предполагается, что это свойство
                                        // больше нигде не используется
        var nextid = 1; // Начальное значение для id

}()); // Вызвать функцию-обертку, чтобы выполнить программный код
```

9.8.2. Определение неизменяемых классов

Помимо возможности делать свойства неперечислимыми, стандарт ECMAScript 5 позволяет делать свойства доступными только для чтения, что может быть довольно удобно при создании классов, экземпляры которых не должны изменяться. В примере 9.18 приводится неизменяемая версия класса `Range`, который использует эту возможность, применяя функции `Object.defineProperties()` и `Object.create()`. Кроме того, функция `Object.defineProperties()` используется в нем также для добавления свойств в объект-прототип класса, что делает методы экземпляров недоступными для перечисления, подобно методам встроенных классов. Но и это еще не все: определяемые в примере методы экземпляров создаются доступными только для чтения и не могут быть удалены, что исключает возможность динамического изменения класса. Наконец, в примере 9.18 использован один интересный трюк – при вызове без ключевого слова `new` функция-конструктор класса действует как фабричная функция.

Пример 9.18. Неизменяемый класс со свойствами и методами, доступными только для чтения

```
// Эта функция может работать и без ключевого слова 'new': она одновременно
// является и конструктором, и фабричной функцией
function Range(from,to) {
    // Дескрипторы свойств from и to, доступных только для чтения.
    var props = {
        from: {value:from, enumerable:true,writable:false,configurable:false},
        to: {value:to, enumerable:true, writable:false, configurable:false}
    };

    if (this instanceof Range)                   // Если вызвана как конструктор
        Object.defineProperties(this, props);    // Определить свойства
    else                                         // Иначе как фабричная функция
        return Object.create(Range.prototype,    // Создать и вернуть новый
                             props);             // объект Range со свойствами
}

// Если добавлять свойства в объект Range.prototype тем же способом, можно будет
// определить атрибуты этих свойств. Поскольку мы не указываем атрибуты enumerable,
// writable и configurable, они по умолчанию получают значение false.
```

```
Object.defineProperties(Range.prototype, {
    includes: {
        value: function(x) { return this.from <= x && x <= this.to; }
    },
    foreach: {
        value: function(f) {
            for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
        }
    },
    toString: {
        value: function() { return "(" + this.from + "..." + this.to + ")"; }
    }
});
```

Для определения неизменяемых и неперечислимых свойств в примере 9.18 используются функции `Object.defineProperties()` и `Object.create()`. Они предоставляют широкие возможности, но необходимость определять для них объекты дескрипторов свойств может сделать программный код более сложным для чтения. Чтобы избежать этого, можно определить вспомогательные функции для изменения атрибутов свойств, которые уже были определены. Две такие вспомогательные функции демонстрируются в примере 9.19.

Пример 9.19. Вспомогательные функции для работы с дескрипторами свойств

```
// Делает указанные (или все) свойства объекта o
// недоступным для записи и настройки.
function freezeProps(o) {
    var props = (arguments.length == 1)              // Если один аргумент,
        ? Object.getOwnPropertyNames(o)              // изменить все свойства,
        : Array.prototype.splice.call(arguments, 1); // иначе только указанные
    props.forEach(function(n) {              // Делает каждое свойство ненастраиваемым
                                             // и доступным только для чтения
        // Пропустить ненастраиваемые свойства
        if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
        Object.defineProperty(o, n, { writable: false, configurable: false });
    });
    return o; // Чтобы можно было продолжить работу с объектом o
}

// Делает неперечислимыми указанные (или все) свойства объекта o,
// если они доступны для настройки.
function hideProps(o) {
    var props = (arguments.length == 1)              // Если один аргумент,
        ? Object.getOwnPropertyNames(o)              // изменить все свойства,
        : Array.prototype.splice.call(arguments, 1); // иначе только указанные
    props.forEach(function(n) {                      // Скрыть каждое от цикла for/in
        // Пропустить ненастраиваемые свойства
        if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
        Object.defineProperty(o, n, { enumerable: false });
    });
    return o;
}
```

Функции `Object.defineProperty()` и `Object.defineProperties()` могут использоваться и для создания новых свойств, и для изменения атрибутов уже существующих свойств. При создании новых свойств все опущенные атрибуты по умолчанию принимают значение `false`. Однако при изменении атрибутов уже существующих свойств опущенные атрибуты не изменяются. Например, в функции `hideProps()` выше указывается только атрибут `enumerable`, потому что функция должна изменять только его.

С помощью этих двух функций можно писать определения классов с использованием преимуществ ECMAScript 5, без существенного изменения привычного стиля определения классов. В примере 9.20 приводится определение неизменяемого класса `Range`, в котором используются наши вспомогательные функции.

Пример 9.20. Более простое определение неизменяемого класса

```
function Range(from, to) {      // Конструктор неизменяемого класса Range
    this.from = from;
    this.to = to;
    freezeProps(this);          // Сделать свойства неизменяемыми
}

Range.prototype = hideProps({  // Определить неперечислимые свойства прототипа
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.3. Сокрытие данных объекта

В разделе 9.6.6 и в примере 9.10 было показано, как можно использовать переменные и аргументы функции-конструктора для сокрытия данных объекта, создаваемого этим конструктором. Недостаток этого приема заключается в том, что в ECMAScript 3 допускается возможность замещения методов доступа к этим данным. Стандарт ECMAScript 5 позволяет обеспечить более надежное сокрытие частных данных за счет определения методов доступа к свойствам, которые не могут быть удалены. Этот способ демонстрируется в примере 9.21.

Пример 9.21. Класс Range со строго инкапсулированными границами

```
// Эта версия класса Range является изменяемой, но она следит за своими
// границами, обеспечивая выполнение условия from <= to.
function Range(from, to) {
    // Проверить соблюдение условия при создании
    if (from > to) throw new Error("Range: значение from должно быть <= to");

    // Определение методов доступа, которые следят за соблюдением условия
    function getFrom() { return from; }
    function getTo() { return to; }
    function setFrom(f) { // Не позволяет устанавливать значение from > to
        if (f <= to) from = f;
        else throw new Error("Range: значение from должно быть <= to");
    }
    function setTo(t) {   // Не позволяет устанавливать значение to < from
        if (t >= from) to = t;
        else throw new Error("Range: значение to должно быть >= from");
    }
    // Создать перечислимые, ненастраиваемые свойства с методами доступа
    Object.defineProperties(this, {
        from: {get:getFrom, set:setFrom, enumerable:true, configurable:false},
        to: { get: getTo, set: setTo, enumerable:true, configurable:false }
    });
}

// Настройка объекта-прототипа осталась такой же, как и в предыдущих примерах.
// Обращение к методам экземпляров чтения свойств from и to выполняется так,
// как если бы они были простыми свойствами.
Range.prototype = hideProps({
    constructor: Range,
    includes: function(x) { return this.from <= x && x <= this.to; },
    foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++) f(x);},
    toString: function() { return "(" + this.from + "..." + this.to + ")"; }
});
```

9.8.4. Предотвращение расширения класса

Возможность расширения классов за счет добавления новых методов в объект-прототип обычно рассматривается как характерная особенность языка JavaScript. Стандарт ECMAScript 5 позволяет при желании предотвратить такую возможность. Функция `Object.preventExtensions()` делает объект нерасширяемым (раздел 6.8.3) – в такой объект невозможно добавить новые свойства. Функция `Object.seal()` идет еще дальше: она не только предотвращает добавление новых свойств, но и делает все имеющиеся свойства ненастраиваемыми, предотвращая возможность их удаления. (Однако ненастраиваемое свойство по-прежнему может быть доступно для записи и по-прежнему может быть преобразовано в свойство, доступное только для чтения.) Чтобы предотвратить возможность расширения объекта `Object.prototype`, можно просто записать:

```
Object.seal(Object.prototype);
```

Другая динамическая особенность языка JavaScript – возможность замены методов объекта:

```
var original_sort_method = Array.prototype.sort;
Array.prototype.sort = function() {
    var start = new Date();
    original_sort_method.apply(this, arguments);
    var end = new Date();
    console.log("Сортировка массива заняла " + (end - start) +
                " миллисекунд.");
};
```

Предотвратить такую замену можно, объявив методы экземпляров доступными только для чтения. Сделать это можно с помощью вспомогательной функции `freezeProps()`, объявленной выше. Другой способ добиться этого эффекта заключается в использовании функции `Object.freeze()`, которая выполняет те же действия, что и функция `Object.seal()`, и дополнительно делает все свойства ненастраиваемыми и доступными только для чтения.

Свойства, доступные только для чтения, обладают одной особенностью, о которой необходимо помнить при работе с классами. Если объект `o` наследует свойство p, доступное только для чтения, попытка присвоить значение свойству o.p будет завершаться неудачей без создания нового свойства в объекте o. Если потребуется переопределить унаследованное свойство, доступное только для чтения, можно воспользоваться функциями Object.defineProperty(), Object.defineProperties() или Object.create(), чтобы создать новое свойство. Это означает, что, когда методы экземпляров класса делаются доступными только для чтения, это существенно осложняет возможность их переопределения в подклассах.

На практике обычно не требуется блокировать возможность изменения объектов-прототипов таким способом, но в некоторых случаях предотвращение расширения объектов может оказаться полезным. Вспомните фабричную функцию enumeration() из примера 9.7. Она сохраняет все экземпляры перечислений в свойствах объекта-прототипа и в свойстве-массиве values конструктора. Эти свойства и массив играют роль официального перечня экземпляров перечислений, и их определенно имеет смысл зафиксировать, чтобы исключить возможность добавления новых экземпляров и изменения или удаления существующих. Для этого достаточно добавить в функцию enumeration() следующие строки:

```
Object.freeze(enumeration.values);
Object.freeze(enumeration);
```

Обратите внимание, что применение функции Object.freeze() к типу перечисления исключает возможность использования свойства objectId, как было показано в примере 9.17. Решение этой проблемы состоит в том, чтобы прочитать значение свойства objectId (вызвать соответствующий метод чтения и установить внутреннее свойство) перечисления только один раз, перед тем как его зафиксировать.

9.8.5. Подклассы и ECMAScript 5

В примере 9.22 демонстрируется порядок создания подклассов с использованием возможностей ECMAScript 5. В нем определяется класс stringSet, наследующий класс AbstractWritableSet из примера 9.16. Основная особенность этого примера заключается в использовании функции Object.create() для создания объекта-прототипа, наследующего прототип суперкласса, и в определении свойств вновь созданного объекта. Как уже отмечалось выше, основная сложность этого подхода заключается в необходимости использовать неудобные дескрипторы свойств.

Другой интересной особенностью этого примера является передача значения null функции Object.create() при создании объекта, не наследующего ничего. Этот объект используется для хранения элементов множества, а тот факт, что он не имеет прототипа, позволяет вместо метода hasOwnProperty() использовать оператор in.

Пример 9.22. StringSet: определение подкласса множества с использованием ECMAScript 5

```
function StringSet() {
    this.set = Object.create(null); // Создать объект без прототипа
    this.n = 0;
    this.add.apply(this, arguments);
}

// Обратите внимание, что Object.create позволяет обеспечить наследование
// прототипа суперкласса и определить методы за счет единственного вызова.
// Поскольку при создании свойств мы не указываем значения атрибутов writable,
// enumerable и configurable, они по умолчанию получают значение false.
// Доступность методов только для чтения усложняет их переопределение в подклассах.
StringSet.prototype = Object.create(AbstractWritableSet.prototype, {
    constructor: { value: StringSet },
    contains: { value: function(x) { return x in this.set; } },
    size: { value: function(x) { return this.n; } },
    foreach: { value: function(f,c) { Object.keys(this.set).forEach(f,c); } },
    add: {
        value: function() {
            for(var i = 0; i < arguments.length; i++) {
                if (!(arguments[i] in this.set)) {
                    this.set[arguments[i]] = true;
                    this.n++;
                }
            }
            return this;
        }
    },
    remove: {
        value: function() {
            for(var i = 0; i < arguments.length; i++) {
                if (arguments[i] in this.set) {
                    delete this.set[arguments[i]];
                    this.n--;
                }
            }
            return this;
        }
    }
});
```

9.8.6. Дескрипторы свойств

В разделе 6.7 дается описание дескрипторов свойств, введенных стандартом ECMAScript 5, но там отсутствуют примеры, демонстрирующие различные случаи их использования. Мы завершим этот раздел, посвященный особенностям ECMAScript 5, расширенным примером, демонстрирующим многие операции со свойствами, допустимые в ECMAScript 5. Программный код в примере 9.23 добавляет в `Object.prototype` метод `properties()` (разумеется, недоступный для перечисления). Значение, возвращаемое этим методом, является объектом, представляющим список свойств и обладающим полезными методами для отображения свойств и атрибутов (которые могут пригодиться при отладке). Его можно использовать для получения дескрипторов свойств (на случай, если потребуется реализовать копирование свойств вместе с их атрибутами) и для установки атрибутов свойств (благодаря чему он может использоваться как альтернатива функциям `hideProps()` и `freezeProps()`, объявленным ранее). Этот единственный пример демонстрирует большинство особенностей свойств в ECMAScript 5, а также применение методики модульного программирования, о которой будет рассказываться в следующем разделе.

Пример 9.23. Особенности свойств в ECMAScript 5

```
/*
 * Определяет метод properties() в Object.prototype, возвращающий объект, который
 * представляет указанные свойства объекта, относительно которого был вызван метод
 * (или все собственные свойства объекта, если метод был вызван без аргументов).
 * Возвращаемый объект имеет четыре полезных метода:
 * toString(), descriptors(), hide() и show().
 */
(function namespace() { // Обернуть все в частную область видимости функции

    // Эта функция будет превращена в метод всех объектов
    function properties() {
        var names;                 // Массив имен свойств
        if (arguments.length == 0) // Все собственные свойства объекта this
            names = Object.getOwnPropertyNames(this);
        else if (arguments.length == 1 && Array.isArray(arguments[0]))
            names = arguments[0];  // Или массив указанных свойств
        else                       // Или имена в списке аргументов
            names = Array.prototype.splice.call(arguments, 0);

        // Вернуть новый объект Properties, представляющий указанные свойства
        return new Properties(this, names);
    }

    // Делает эту функцию новым, неперечислимым свойством Object.prototype.
    // Это единственное значение, экспортируемое из частной области видимости функции.
    Object.defineProperty(Object.prototype, "properties", {
        value: properties,
        enumerable: false, writable: true, configurable: true
    });

    // Следующая функция-конструктор вызывается функцией properties().
    // Класс Properties представляет множество свойств объекта.
    function Properties(o, names) {
        this.o = o;          // Объект, которому принадлежат свойства
        this.names = names; // Имена свойств
    }

    // Делает неперечислимыми свойства, представленные объектом this
    Properties.prototype.hide = function() {
        var o = this.o, hidden = { enumerable: false };
        this.names.forEach(function(n) {
                               if (o.hasOwnProperty(n))
                                   Object.defineProperty(o, n, hidden);
                           });
        return this;
    };

    // Делает свойства ненастраиваемыми и доступными только для чтения
    Properties.prototype.freeze = function() {
        var o = this.o, frozen = { writable: false, configurable: false };
        this.names.forEach(function(n) {
                               if (o.hasOwnProperty(n))
                                   Object.defineProperty(o, n, frozen);
                           });
        return this;
    };

    // Возвращает объект, отображающий имена свойств в дескрипторы.
    // Может использоваться для реализации копирования свойств вместе с их атрибутами:
    //   Object.defineProperties(dest, src.properties().descriptors());
    Properties.prototype.descriptors = function() {
        var o = this.o, desc = {};
        this.names.forEach(function(n) {
                              if (!o.hasOwnProperty(n)) return;
                              desc[n] = Object.getOwnPropertyDescriptor(o,n);
                          });
        return desc;
    };

    // Возвращает отформатированный список свойств, в котором перечислены имена,
    // значения и атрибуты свойств. Термин "permanent" используется для обозначения
    // ненастраиваемых свойств, "readonly" - для обозначения свойств, не доступных
    // для записи, и "hidden" - для обозначения неперечислимых свойств.
    // Обычные перечислимые, доступные для записи и настраиваемые свойства
    // указываются в списке без атрибутов.
    Properties.prototype.toString = function() {
        var o = this.o; // Используется во вложенных функциях ниже
        var lines = this.names.map(nameToString);
        return "{\n  " + lines.join(",\n  ") + "\n}";

        function nameToString(n) {
            var s = "", desc = Object.getOwnPropertyDescriptor(o, n);
            if (!desc) return "nonexistent " + n + ": undefined";
            if (!desc.configurable) s += "permanent ";
            if ((desc.get && !desc.set) || !desc.writable) s += "readonly ";
            if (!desc.enumerable) s += "hidden ";
            if (desc.get || desc.set) s += "accessor " + n
            else s += n + ": " + ((typeof desc.value==="function")?"function"
                                                                 :desc.value);
            return s;
        }
    };

    // Наконец, сделать методы экземпляров объекта-прототипа, объявленного
    // выше, неперечислимыми, с помощью методов, объявленных здесь.
    Properties.prototype.properties().hide();
}()); // Вызвать вмещающую функцию сразу после ее определения.
```

9.9. Модули

Важной причиной организации программного кода в классы является стремление придать этому программному коду *модульную структуру* и обеспечить возможность повторного его использования в различных ситуациях. Однако не только классы могут использоваться для организации модульной структуры. Обычно модулем считается один отдельный файл с программным кодом JavaScript. Файл модуля может содержать определение класса, множество родственных классов, библиотеку вспомогательных функций или просто выполняемый сценарий. Модулем может быть любой фрагмент программного кода на языке JavaScript при

условии, что он имеет модульную структуру. В языке JavaScript отсутствуют какие-либо синтаксические конструкции для работы с модулями (впрочем, в нем имеются зарезервированные ключевые слова `imports` и `exports`, поэтому такие конструкции могут появиться в будущих версиях языка), поэтому написание модулей в языке JavaScript в значительной степени является вопросом следования соглашениям оформления программного кода.

Многие библиотеки и клиентские фреймворки JavaScript включают собственные инструменты поддержки модулей. Например, библиотеки Dojo и Google Closure определяют функции `provide()` и `require()` для объявления и загрузки модулей. А в рамках проекта CommonJS по стандартизации серверного JavaScript (*http://commonjs.org*) разработана спецификация, определяющая модули, в которой также используется функция `require()`. Подобные инструменты поддержки модулей часто берут на себя такие функции, как загрузка модулей и управление зависимостями, но их обсуждение выходит за рамки этой дискуссии. Если вы пользуетесь одним из таких фреймворков, то вам следует использовать и определять модули, следуя соглашениям, принятым в этом фреймворке. А в этом разделе мы обсудим лишь самые простые соглашения.

Цель модульной организации заключается в том, чтобы обеспечить возможность сборки больших программ из фрагментов программного кода, полученных из различных источников, и нормальную работу всех этих фрагментов, включая программный код, авторы которого не предусматривали подобную возможность. Для безошибочной работы модули должны стремиться избегать внесения изменений в глобальную среду выполнения, чтобы последующие модули могли выполняться в нетронутом (или почти не тронутом) окружении. С практической точки зрения это означает, что модули должны до минимума уменьшить количество определяемых ими глобальных имен – в идеале каждый модуль должен определять не более одного имени. В следующих подразделах описываются простые способы достижения этой цели. Вы увидите, что создание модулей в языке JavaScript не связано с какими-либо сложностями: мы уже видели примеры использования приемов, описываемых здесь, на протяжении этой книги.

9.9.1. Объекты как пространства имен

Один из способов обойтись в модуле без создания глобальных переменных заключается в том, чтобы создать объект и использовать его как пространство имен. Вместо того чтобы создавать глобальные функции и переменные, их можно сохранять в свойствах объекта (на который может ссылаться глобальная переменная). Рассмотрим в качестве примера класс `Set` из примера 9.6. Он определяет единственную глобальную функцию-конструктор `Set`. Он определяет различные методы экземпляров, но сохраняет их как свойства объекта `Set.prototype`, благодаря чему они уже не являются глобальными. В этом примере также определяется вспомогательная функция `_v2s()`, но она также сохраняется как свойство класса `Set`.

Далее рассмотрим пример 9.16. Этот пример объявляет несколько абстрактных и конкретных классов множеств. Для каждого класса создается единственное глобальное имя, но модуль целиком (файл с программным кодом) определяет довольно много глобальных имен. С точки зрения сохранения чистоты глобального пространства имен было бы лучше, если бы модуль с классами множеств определял единственное глобальное имя:

```
var sets = {};
```

Объект `sets` мог бы играть роль пространства имен модуля, а каждый из классов множеств определялся бы как свойство этого объекта:

```
sets.SingletonSet = sets.AbstractEnumerableSet.extend(...);
```

Когда возникла бы потребность использовать класс, объявленный таким способом, мы могли бы просто добавлять пространство имен при ссылке на конструктор:

```
var s = new sets.SingletonSet(1);
```

Автор модуля не может заранее знать, с какими другими модулями будет использоваться его модуль, поэтому он должен принять все меры против конфликтов, используя подобные пространства имен. Однако программист, использующий модуль, знает, какие модули он использует и какие имена в них определяются. Этот программист не обязан использовать имеющиеся пространства имен ограниченно и может *импортировать* часто используемые значения в глобальное пространство имен. Программист, который собирается часто использовать класс `Set` из пространства имен `sets`, мог бы импортировать класс, как показано ниже:

```
var Set = sets.Set;      // Импортировать Set в глобальное пространство имен
var s = new Set(1,2,3); // Теперь его можно использовать без префикса sets.
```

Иногда авторы модулей используют глубоко вложенные пространства имен. Если модуль с определениями классов множеств является частью обширной коллекции модулей, для него может использоваться пространство имен `collections.sets`, а сам модуль может начинаться со следующих определений:

```
var collections;        // Объявить (или повторно объявить) глобальную переменную
if (!collections)       // Если объект еще не существует
    collections = {};   // Создать объект пространства имен верхнего уровня
collections.sets = {}   // И внутри него создать пространство имен sets.
// Теперь определить классы множеств внутри collections.sets
collections.sets.AbstractSet = function() { ... }
```

Иногда пространство имен верхнего уровня используется для идентификации разработчика или организации – автора модуля и для предотвращения конфликтов между пространствами имен. Например, библиотека Google Closure определяет свой класс `Set` в пространстве имен `goog.structs`. Для определения глобально уникальных префиксов, которые едва ли будут использоваться другими авторами модулей, индивидуальные разработчики могут использовать компоненты доменного имени. Поскольку мой веб-сайт имеет имя *davidflanagan.com*, я мог бы поместить свой модуль с классами множеств в пространство имен `com.davidflanagan.collections.sets`.

При таких длинных именах для любого пользователя вашего модуля операция импортирования становится особенно важной. Однако, вместо того чтобы импортировать имена отдельных классов, программист мог бы импортировать в глобальное пространство имен весь модуль целиком:

```
var sets = com.davidflanagan.collections.sets;
```

В соответствии с соглашениями имя файла модуля должно совпадать с его пространством имен. Модуль sets должен храниться в файле с именем *sets.js*. Если модуль использует пространство имен collections.sets, то этот файл должен храниться в каталоге *collections/* (этот каталог мог бы также включать файл *maps.js*). А модуль, использующий пространство имен com.davidflanagan.collections.sets, должен храниться в файле *com/davidflanagan/collections/sets.js*.

9.9.2. Область видимости функции как частное пространство имен

Модули имеют экспортируемый ими общедоступный прикладной интерфейс (API): это функции, классы, свойства и методы, предназначенные для использования другими программистами. Однако зачастую для внутренних нужд модуля требуются дополнительные функции или методы, которые не предназначены для использования за пределами модуля. Примером может служить функция Set._v2s() из примера 9.6 – для нас было бы нежелательно, чтобы пользователи класса Set вызывали эту функцию, поэтому было бы неплохо сделать ее недоступной извне.

Этого можно добиться, определив модуль (в данном случае класс Set) внутри функции. Как описывалось в разделе 8.5, переменные и функции, объявленные внутри другой функции, являются локальными по отношению к этой функции и недоступны извне. Таким образом, область видимости функции (называемой иногда «функцией модуля») можно использовать как частное пространство имен модуля. Пример 9.24 демонстрирует, как это может выглядеть применительно к нашему классу Set.

Пример 9.24. Класс Set внутри функции модуля

```
// Объявляет глобальную переменную Set и присваивает ей значение, возвращаемое
// функцией. Круглые скобки, окружающие объявление функции, свидетельствуют о том,
// что функция будет вызвана сразу после ее объявления и что присваивается значение,
// возвращаемое функцией, а не сама функция. Обратите внимание, что это выражение
// определения функции, а не инструкция, поэтому наличие имени "invocation"
// не вызывает создание глобальной переменной.
var Set = (function invocation() {

    function Set() { // Эта функция-конструктор - локальная переменная.
        this.values = {}; // Свойство для хранения множества
        this.n = 0;       // Количество значений в множестве
        this.add.apply(this, arguments); // Все аргументы являются значениями,
    }                                    // добавляемыми в множество

    // Далее следуют определения методов в Set.prototype.
    // Для экономии места программный код опущен
    Set.prototype.contains = function(value) {
        // Обратите внимание, что v2s() вызывается без префикса Set._v2s()
        return this.values.hasOwnProperty(v2s(value));
    };
Set.prototype.size = function() { return this.n; };
Set.prototype.add = function() { /* ... */ };
Set.prototype.remove = function() { /* ... */ };
Set.prototype.foreach = function(f, context) { /* ... */ };
```

```
    // Далее следуют вспомогательные функции и переменные, используемые
    // методами выше. Они не являются частью общедоступного API модуля и скрыты
    // в области видимости функции, благодаря чему не требуется объявлять их как
    // свойства класса Set или использовать символ подчеркивания в качестве префикса.
    function v2s(val) { /* ... */ }
    function objectId(o) { /* ... */ }
    var nextId = 1;

    // Общедоступным API этого модуля является функция-конструктор Set().
    // Нам необходимо экспортировать эту функцию за пределы частного
    // пространства имен, чтобы ее можно было использовать за ее пределами.
    // В данном случае конструктор экспортируется за счет передачи его
    // в виде возвращаемого значения. Он становится присваиваемым значением
    // в выражении в первой строке выше.
    return Set;
}()); // Вызвать функцию сразу после ее объявления.
```

Обратите внимание, что такой прием вызова функции сразу после ее определения является характерным для языка JavaScript. Программный код, выполняемый в частном пространстве имен, предваряется текстом «(function() {» и завершается «}());». Открывающая круглая скобка в начале сообщает интерпретатору, что это выражение определения функции, а не инструкция, поэтому в префикс можно добавить любое имя функции, поясняющее ее назначение. В примере 9.24 было использовано имя «invocation», чтобы подчеркнуть, что функция вызывается сразу же после ее объявления. Точно так же можно было бы использовать имя «namespace», чтобы подчеркнуть, что функция играет роль пространства имен.

После того как модуль окажется заперт внутри функции, ему необходим некоторый способ экспортировать общедоступный API для использования за пределами функции модуля. В примере 9.24 функция модуля возвращает конструктор, который тут же присваивается глобальной переменной. Сам факт возврата значения из функции ясно говорит о том, что оно экспортируется за пределы области видимости функции. Модули, определяющие более одного элемента API, могут возвращать объект пространства имен. Для нашего модуля с классами множеств можно было бы написать такой программный код:

```
// Создает единственную глобальную переменную, хранящую все модули,
// имеющие отношение к коллекциям
var collections;
if (!collections) collections = {};

// Теперь определить модуль sets
collections.sets = (function namespace() {
    // Здесь находятся определения различных классов множеств,
    // использующих локальные переменные и функции
    // ... Большая часть программного кода опущена...
    // Экспортировать API в виде возвращаемого объекта пространства имен
    return {
        // Экспортируемое имя свойства : имя локальной переменной
        AbstractSet: AbstractSet,
        NotSet: NotSet,
        AbstractEnumerableSet: AbstractEnumerableSet,
        SingletonSet: SingletonSet,
        AbstractWritableSet: AbstractWritableSet,
        ArraySet: ArraySet
    };
}());
```

Можно предложить похожий прием, определив функцию модуля как конструктор, который будет вызываться с ключевым словом `new` и экспортировать значения за счет их присваивания:

```
var collections;
if (!collections) collections = {};
collections.sets = (new function namespace() {
    // ... Большая часть программного кода опущена ...
    // Экспортировать API в обьекте this
    this.AbstractSet = AbstractSet;
    this.NotSet = NotSet;  // И так далее....
    // Обратите внимание на отсутствие возвращаемого значения.
}());
```

Если объект глобального пространства имен уже определен, функция модуля может просто присваивать значения свойствам этого объекта и вообще ничего не возвращать:

```
var collections;
if (!collections) collections = {};
collections.sets = {};
(function namespace() {
    // ... Большая часть программного кода опущена ...
    // Экспортировать общедоступный API в объект пространства имен, созданный выше
    collections.sets.AbstractSet = AbstractSet;
    collections.sets.NotSet = NotSet; // И так далее...
    // Инструкция return не требуется, потому что экспортирование выполняется выше.
}());
```

Фреймворки, реализующие инструменты загрузки модулей, могут предусматривать собственные методы экспортирования API модулей. Внутри модуля может определяться функция `provides()`, которая выполняет регистрацию его API, или объект `exports`, в котором модуль должен сохранять свой API. Пока в языке JavaScript отсутствуют инструменты управления модулями, вам придется использовать средства создания и экспортирования модулей, которые лучше подходят для используемой вами библиотеки инструментов.

10

Шаблоны и регулярные выражения

Регулярное выражение – это объект, описывающий символьный шаблон. Класс `RegExp` в JavaScript представляет регулярные выражения, а объекты классов `String` и `RegExp` определяют методы, использующие регулярные выражения для выполнения поиска по шаблону и операций поиска в тексте с заменой. Грамматика регулярных выражений в языке JavaScript содержит достаточно полное подмножество синтаксиса регулярных выражений, используемого в языке Perl 5, поэтому, если вы имеете опыт работы с языком Perl, то вы без труда сможете описывать шаблоны в программах на языке JavaScript.[1]

Эта глава начинается с определения синтаксиса, посредством которого в регулярных выражениях описываются текстовые шаблоны. Затем мы перейдем к описанию тех методов классов `String` и `RegExp`, которые используют регулярные выражения.

10.1. Определение регулярных выражений

В JavaScript регулярные выражения представлены объектами `RegExp`. Объекты `RegExp` могут быть созданы посредством конструктора `RegExp()`, но чаще они создаются с помощью специального синтаксиса литералов. Так же как строковые литералы задаются в виде символов, заключенных в кавычки, литералы регулярных выражений задаются в виде символов, заключенных в пару символов слэша (/). Таким образом, JavaScript-код может содержать строки, похожие на эту:

```
var pattern = /s$/;
```

---

[1] В число особенностей регулярных выражений языка Perl, которые не поддерживаются в ECMAScript, входят флаги `s` (однострочный режим) и `x` (расширенный синтаксис); управляющие последовательности `\a`, `\e`, `\l`, `\u`, `\L`, `\U`, `\E`, `\Q`, `\A`, `\Z`, `\z` и `\G`; якорь `(?<=` позитивной ретроспективной проверки и якорь `(?<!` негативной ретроспективной проверки; комментарии`(?#` и другие расширенные конструкции, начинающиеся с `(?`.

Эта строка создает новый объект `RegExp` и присваивает его переменной `pattern`. Данный объект `RegExp` ищет любые строки, заканчивающиеся символом «s». Это же регулярное выражение может быть определено с помощью конструктора `RegExp()`:

```
var pattern = new RegExp("s$");
```

Спецификация шаблона регулярного выражения состоит из последовательности символов. Большая часть символов, включая все алфавитно-цифровые, буквально описывают символы, которые должны присутствовать. То есть регулярное выражение `/java/` совпадает со всеми строками, содержащими подстроку «java». Другие символы в регулярных выражениях не предназначены для поиска их точных эквивалентов, а имеют особое значение. Например, регулярное выражение `/s$/` содержит два символа. Первый символ, `s`, обозначает поиск буквального символа. Второй, `$`, – это специальный метасимвол, обозначающий конец строки. Таким образом, это регулярное выражение соответствует любой строке, заканчивающейся символом `s`.

В следующих разделах описаны различные символы и метасимволы, используемые в регулярных выражениях в языке JavaScript.

Литералы RegExp и создание объектов

Литералы простых типов, таких как строки и числа, интерпретируются как одни и те же значения, где бы они ни встретились в программе. Литералы объектов (или инициализаторы), такие как `{}` и `[]`, каждый раз создают новые объекты. Если поместить инструкцию `var a = []` в тело цикла, например, в каждой итерации цикла будет создаваться новый пустой массив.

Литералы регулярных выражений – особый случай. Спецификация ECMAScript 3 утверждает, что литерал `RegExp` преобразуется в объект `RegExp` в ходе синтаксического анализа программного кода и каждый раз, когда интерпретатор встречает литерал `RegExp`, он возвращает один и тот же объект. Спецификация ECMAScript 5 изменила это положение вещей и требует, чтобы всякий раз, когда в программе встречается литерал `RegExp`, возвращался бы новый объект. Реализация в броузере IE всегда соответствовала поведению, соответствующему ECMAScript 5, и большинство современных броузеров также перешли на новую реализацию, раньше, чем полностью реализовали новый стандарт.

10.1.1. Символы литералов

Как отмечалось ранее, все алфавитные символы и цифры в регулярных выражениях соответствуют сами себе. Синтаксис регулярных выражений в JavaScript также поддерживает возможность указывать некоторые неалфавитные символы с помощью управляющих последовательностей, начинающихся с символа обратного слэша (\). Например, последовательность `\n` соответствует символу перевода строки. Эти символы перечислены в табл. 10.1.

Таблица 10.1. Символы литералов в регулярных выражениях

Символ	**Соответствие**
Алфавитно-цифровые символы	Соответствуют сами себе
`\0`	Символ NUL (`\u0000`)
`\t`	Табуляция (`\u0009`)
`\n`	Перевод строки (`\u000A`)
`\v`	Вертикальная табуляция (`\u000B`)
`\f`	Перевод страницы (`\u000C`)
`\r`	Возврат каретки (`\u000D`)
`\x`*nn*	Символ из набора Latin, задаваемый шестнадцатеричным числом *nn*; например, `\x0A` – это то же самое, что `\n`
`\u`*xxxx*	Unicode-символ, заданный шестнадцатеричным числом *xxxx*; например, `\u0009` – это то же самое, что `\t`
`\c`*X*	Управляющий символ `^`*X*; например, последовательность `\cJ` эквивалентна символу перевода строки `\n`

Некоторые знаки препинания имеют в регулярных выражениях особый смысл:

`^ $ . * + ? = ! : | \ / ( ) [ ] { }`

Значение этих символов раскрывается в последующих разделах. Некоторые из них имеют специальный смысл только в определенных контекстах регулярных выражений, а в других контекстах трактуются буквально. Однако, как правило, чтобы включить какой-либо из этих символов в регулярное выражение буквально, необходимо поместить перед ним символ обратного слэша. Другие символы, такие как кавычки и `@`, не имеют специального значения и просто соответствуют в регулярных выражениях самим себе.

Если вы не можете точно вспомнить, каким из символов должен предшествовать символ `\`, можете спокойно помещать обратный слэш перед любым из символов. Однако имейте в виду, что многие буквы и цифры вместе с символом слэша обретают специальное значение, поэтому тем буквам и цифрам, которые вы ищете буквально, не должен предшествовать символ `\`. Чтобы включить в регулярное выражение сам символ обратного слэша, перед ним, очевидно, следует поместить другой символ обратного слэша. Например, следующее регулярное выражение соответствует любой строке, содержащей символ обратного слэша: `/\\/`.

10.1.2. Классы символов

Отдельные символы литералов могут объединяться в *классы символов* путем помещения их в квадратные скобки. Класс символов соответствует любому символу, содержащемуся в этом классе. Следовательно, регулярное выражение `/[abc]/` соответствует одному из символов `a`, `b` или `c`. Могут также определяться классы символов с отрицанием, соответствующие любому символу, кроме тех, которые указаны в скобках. Класс символов с отрицанием задается символом `^` в качестве первого символа, следующего за левой скобкой. Регулярное выражение `/[^abc]/`

соответствует любому символу, отличному от a, b или c. В классах символов диапазон символов может задаваться при помощи дефиса. Поиск всех символов латинского алфавита в нижнем регистре осуществляется посредством выражения /[a-z]/, а любую букву или цифру из набора символов Latin можно найти при помощи выражения /[a-zA-Z0-9]/.

Некоторые классы символов используются особенно часто, поэтому синтаксис регулярных выражений в JavaScript включает специальные символы и управляющие (escape) последовательности для их обозначения. Так, \s соответствует символам пробела, табуляции и любым пробельным символам из набора Unicode, а \S – любым символам, *не* являющимся пробельными символами из набора Unicode. В табл. 10.2 приводится перечень этих спецсимволов и синтаксиса классов символов. (Обратите внимание, что некоторые из управляющих последовательностей классов символов соответствуют только ASCII-символам и не расширены для работы с Unicode-символами. Можно явно определить собственные классы Unicode-символов, например, выражение /[\u0400-\04FF]/ соответствует любому символу кириллицы.)

Таблица 10.2. Классы символов регулярных выражений

Символ	Соответствие
[...]	Любой из символов, указанных в скобках
[^...]	Любой из символов, не указанных в скобках
	Любой символ, кроме перевода строки или другого разделителя Unicode-строки
\w	Любой текстовый ASCII-символ. Эквивалентно [a-zA-Z0-9_]
\W	Любой символ, не являющийся текстовым ASCII-символом. Эквивалентно [^a-zA-Z0-9_]
\s	Любой пробельный символ из набора Unicode
\S	Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S – это не одно и то же
\d	Любые ASCII-цифры. Эквивалентно [0-9]
\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]
[\b]	Литерал символа «забой» (особый случай)

Обратите внимание, что управляющие последовательности специальных символов классов могут находиться в квадратных скобках. \s соответствует любому пробельному символу, а \d соответствует любой цифре, следовательно, /[\s\d]/ соответствует любому пробельному символу или цифре. Обратите внимание на особый случай. Как мы увидим позже, последовательность \b имеет особый смысл. Однако когда она используется в классе символов, то обозначает символ «забой». Поэтому, чтобы обозначить символ «забой» в регулярном выражении буквально, используйте класс символов с одним элементом: /[\b]/.

10.1.3. Повторение

Имея знания синтаксиса регулярных выражений, полученные к настоящему моменту, мы можем описать число из двух цифр как /\d\d/ или из четырех цифр как

/\d\d\d\d/, но не сможем, например, описать число, состоящее из любого количества цифр, или строку из трех букв, за которыми следует необязательная цифра. Эти более сложные шаблоны используют синтаксис регулярных выражений, указывающий, сколько раз может повторяться данный элемент регулярного выражения.

Символы, обозначающие повторение, всегда следуют за шаблоном, к которому они применяются. Некоторые виды повторений используются довольно часто, и для обозначения этих случаев имеются специальные символы. Например, + соответствует одному или нескольким экземплярам предыдущего шаблона. В табл. 10.3 приведена сводка синтаксиса повторений.

Таблица 10.3. Символы повторения в регулярных выражениях

Символ	Значение
{*n*,*m*}	Соответствует предшествующему шаблону, повторенному не менее *n* и не более *m* раз
{*n*,}	Соответствует предшествующему шаблону, повторенному *n* или более раз
{*n*}	Соответствует в точности *n* экземплярам предшествующего шаблона
?	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным. Эквивалентно {0,1}
+	Соответствует одному или более экземплярам предшествующего шаблона. Эквивалентно {1,}
*	Соответствует нулю или более экземплярам предшествующего шаблона. Эквивалентно {0,}

Следующие строки демонстрируют несколько примеров:

```
/\d{2,4}/    // Соответствует числу, содержащему от двух до четырех цифр
/\w{3}\d?/   // Соответствует в точности трем символам слова
             // и одной необязательной цифре
/\s+java\s+/ // Соответствует слову "java" с одним или более пробелами
             // до и после него
/[^(]*/      // Соответствует нулю или более символам, отличным от открывающей круглой
             // скобки
```

Будьте внимательны при использовании символов повторения * и ?. Они могут соответствовать отсутствию указанного перед ними шаблона и, следовательно, отсутствию символов. Например, регулярному выражению /a*/ соответствует строка «bbbb», поскольку в ней нет символа a!

10.1.3.1. «Нежадное» повторение

Символы повторения, перечисленные в табл. 10.3, соответствуют максимально возможному количеству повторений, при котором обеспечивается поиск последующих частей регулярного выражения. Мы говорим, что это – «жадное» повторение. Имеется также возможность реализовать повторение, выполняемое «нежадным» способом. Достаточно указать после символа (или символов) повторения вопросительный знак: ??, +?, *? или даже {1,5}?. Например, регулярное выражение /a+/ соответствует одному или более экземплярам буквы a. Примененное к строке «aaa», оно соответствует всем трем буквам. С другой стороны, выражение /a+?/

соответствует одному или более экземплярам буквы a и выбирает наименее возможное число символов. Примененный к той же строке, этот шаблон соответствует только первой букве a.

«Нежадное» повторение не всегда дает ожидаемый результат. Рассмотрим шаблон /a+b/, соответствующий одному или более символам a, за которыми следует символ b. Применительно к строке «aaab» ему соответствует вся строка. Теперь проверим «нежадную» версию /a+?b/. Можно было бы подумать, что она должна соответствовать символу b, перед которым стоит только один символ a. В случае применения к той же строке «aaab» можно было бы ожидать, что она совпадет с единственным символом a и последним символом b. Однако на самом деле этому шаблону соответствует вся строка, как и в случае «жадной» версии. Дело в том, что поиск по шаблону регулярного выражения выполняется путем нахождения первой позиции в строке, начиная с которой соответствие становится возможным. Так как соответствие возможно, начиная с первого символа строки, более короткие соответствия, начинающиеся с последующих символов, даже не рассматриваются.

10.1.4. Альтернативы, группировка и ссылки

Грамматика регулярных выражений включает специальные символы определения альтернатив, подвыражений группировки и ссылок на предыдущие подвыражения. Символ вертикальной черты | служит для разделения альтернатив. Например, /ab|cd|ef/ соответствует либо строке «ab», либо строке «cd», либо строке «ef», а шаблон /\d{3}|[a-z]{4}/ – либо трем цифрам, либо четырем строчным буквам.

Обратите внимание, что альтернативы обрабатываются слева направо до тех пор, пока не будет найдено соответствие. При обнаружении совпадения с левой альтернативой правая игнорируется, даже если можно добиться «лучшего» соответствия. Поэтому, когда к строке «ab» применяется шаблон /a|ab/, он будет соответствовать только первому символу.

Круглые скобки имеют в регулярных выражениях несколько значений. Одно из них – группировка отдельных элементов в одно подвыражение, так что элементы при использовании специальных символов |, *, +, ? и других рассматриваются как одно целое. Например, шаблон /java(script)?/ соответствует слову «java», за которым следует необязательное слово «script», а /(ab|cd)+|ef)/ соответствует либо строке «ef», либо одному или более повторений одной из строк «ab» или «cd».

Другим применением скобок в регулярных выражениях является определение подшаблонов внутри шаблона. Когда в целевой строке найдено совпадение с регулярным выражением, можно извлечь часть целевой строки, соответствующую любому конкретному подшаблону, заключенному в скобки. (Мы увидим, как получить эти подстроки, далее в этой главе.) Предположим, что требуется отыскать одну или более букв в нижнем регистре, за которыми следует одна или несколько цифр. Для этого можно воспользоваться шаблоном /[a-z]+\d+/. Но предположим также, что нам нужны только цифры в конце каждого соответствия. Если поместить эту часть шаблона в круглые скобки (/[a-z]+(\d+)/), то можно будет извлечь цифры из любых найденных нами соответствий. Как это делается, будет описано ниже.

С этим связано еще одно применение подвыражений в скобках, позволяющее ссылаться на подвыражения из предыдущей части того же регулярного выражения. Это достигается путем указания одной или нескольких цифр после символа \. Цифры ссылаются на позицию подвыражения в скобках внутри регулярного выражения. Например, `\1` ссылается на первое подвыражение, а `\3` – на третье. Обратите внимание, что подвыражения могут быть вложены одно в другое, поэтому при подсчете используется позиция левой скобки. Например, в следующем регулярном выражении ссылка на вложенное подвыражение `([Ss]cript)` будет выглядеть как `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

Ссылка на предыдущее подвыражение указывает *не* на шаблон этого подвыражения, а на найденный текст, соответствующий этому шаблону. Поэтому ссылки могут использоваться для наложения ограничения, выбирающего части строки, содержащие точно такие же символы. Например, следующее регулярное выражение соответствует нулю или более символам внутри одинарных или двойных кавычек. Однако оно не требует, чтобы открывающие и закрывающие кавычки соответствовали друг другу (т. е. чтобы обе кавычки были одинарными или двойными):

```
/['"][^'"]*['"]/
```

Соответствия кавычек мы можем потребовать посредством такой ссылки:

```
/(['"])[^'"]*\1/
```

Здесь `\1` соответствует совпадению с первым подвыражением. В этом примере ссылка налагает ограничение, требующее, чтобы закрывающая кавычка соответствовала открывающей. Это регулярное выражение не допускает присутствия одинарных кавычек внутри двойных, и наоборот. Недопустимо помещать ссылки внутрь классов символов, т. е. мы не можем написать:

```
/(['"])[^\1]*\1/
```

Далее в этой главе мы увидим, что этот вид ссылок на подвыражения представляет собой мощное средство использования регулярных выражений в операциях поиска с заменой.

Возможна также группировка элементов в регулярном выражении без создания нумерованной ссылки на эти элементы. Вместо простой группировки элементов между `(` и `)` начните группу с символов `(?:` и закончите ее символом `)`. Рассмотрим, например, следующий шаблон:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Здесь подвыражение `(?:[Ss]cript)` необходимо только для группировки, чтобы к группе мог быть применен символ повторения `?`. Эти модифицированные скобки не создают ссылку, поэтому в данном регулярном выражении `\2` ссылается на текст, соответствующий шаблону `(fun\w*)`.

В табл. 10.4 приводится перечень операторов выбора из альтернатив, группировки и ссылки в регулярных выражениях.

Таблица 10.4. Символы регулярных выражений выбора из альтернатив, группировки и ссылки

Символ	Значение
|	Альтернатива. Соответствует либо подвыражению слева, либо подвыражению справа.
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с символами *, +, ?, | и т. п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках.
(?:...)	Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе.
\n	Соответствует тем же символам, которые были найдены при сопоставлении с группой с номером *n*. Группы – это подвыражения внутри скобок (возможно, вложенных). Номера группам присваиваются путем подсчета левых скобок слева направо. Группы, сформированные с помощью символов (?:, не нумеруются.

10.1.5. Указание позиции соответствия

Как описывалось ранее, многие элементы регулярного выражения соответствуют одному символу в строке. Например, \s соответствует одному пробельному символу. Другие элементы регулярных выражений соответствуют позициям между символами, а не самим символам. Например, \b соответствует границе слова – границе между \w (текстовый ASCII-символ) и \W (нетекстовый символ) или границе между текстовым ASCII-символом и началом или концом строки.[1] Такие элементы, как \b, не определяют какие-либо символы, которые должны присутствовать в найденной строке, однако они определяют допустимые позиции для проверки соответствия. Иногда эти элементы называются *якорными элементами регулярных выражений*, потому что они закрепляют шаблон за определенной позицией в строке. Чаще других используются такие якорные элементы, как ^ и $, привязывающие шаблоны соответственно к началу и концу строки.

Например, слово «JavaScript», находящееся на отдельной строке, можно найти с помощью регулярного выражения /^JavaScript$/. Чтобы найти отдельное слово «Java» (а не префикс, например в слове «JavaScript»), можно попробовать применить шаблон /\sJava\s/, который требует наличия пробела[2] до и после слова. Но такое решение порождает две проблемы. Во-первых, оно найдет слово «Java», только если оно окружено пробелами с обеих сторон, и не сможет найти его в начале или в конце строки. Во-вторых, когда этот шаблон действительно найдет соответствие, возвращаемая им строка будет содержать ведущие и замыкающие пробелы, а это не совсем то, что нам нужно. Поэтому вместо шаблона, совпадающего с пробельными символами \s, мы воспользуемся шаблоном (или якорем), совпадающим с границами слова \b. Получится следующее выражение: /\bJava\b/. Якорный элемент \B соответствует позиции, не являющейся границей слова.

---

[1] За исключением класса символов (квадратных скобок), где \b соответствует символу «забой».

[2] Точнее, любого пробельного символа. – *Прим. науч. ред.*

То есть шаблону `/\B[Ss]cript/` будут соответствовать слова «JavaScript» и «postscript» и не будут соответствовать слова «script» или «Scripting».

В качестве якорных условий могут также выступать произвольные регулярные выражения. Если поместить выражение между символами `(?=` и `)`, оно превратится в опережающую проверку на совпадение с последующими символами, требующую, чтобы эти символы соответствовали указанному шаблону, но не включались в строку соответствия. Например, чтобы найти совпадение с названием распространенного языка программирования, за которым следует двоеточие, можно воспользоваться выражением `/[Jj]ava([Ss]cript)?(?=\:)/`. Этому шаблону соответствует слово «JavaScript» в строке «JavaScript: The Definitive Guide», но ему не будет соответствовать слово «Java» в строке «Java in a Nutshell», потому что за ним не следует двоеточие.

Если же ввести условие `(?!`, то это будет негативная опережающая проверка на последующие символы, требующая, чтобы следующие символы не соответствовали указанному шаблону. Например, шаблону `/Java(?!Script)([A-Z]\w*)/` соответствует подстрока «Java», за которой следует заглавная буква и любое количество текстовых ASCII-символов при условии, что за подстрокой «Java» не следует подстрока «Script». Он совпадет со строкой «JavaBeans», но не совпадет со строкой «Javanese», совпадет со строкой «JavaScrip», но не совпадет со строками «JavaScript» или «JavaScripter».

В табл. 10.5 приводится перечень якорных символов регулярных выражений.

Таблица 10.5. Якорные символы регулярных выражений

Символ	Значение
`^`	Соответствует началу строкового выражения или началу строки при многострочном поиске.
`$`	Соответствует концу строкового выражения или концу строки при многострочном поиске.
`\b`	Соответствует границе слова, т. е. соответствует позиции между символом `\w` и символом `\W` или между символом `\w` и началом или концом строки. (Однако обратите внимание, что `[\b]` соответствует символу забоя.)
`\B`	Соответствует позиции, не являющейся границей слов.
`(?=`*p*`)`	Позитивная опережающая проверка на последующие символы. Требует, чтобы последующие символы соответствовали шаблону *p*, но не включает эти символы в найденную строку.
`(?!`*p*`)`	Негативная опережающая проверка на последующие символы. Требует, чтобы следующие символы не соответствовали шаблону *p*.

10.1.6. Флаги

И еще один, последний элемент грамматики регулярных выражений. Флаги регулярных выражений задают высокоуровневые правила соответствия шаблонам. В отличие от остальной грамматики регулярных выражений, флаги указываются не между символами слэша, а после второго из них. В языке JavaScript поддерживается три флага. Флаг `i` указывает, что поиск по шаблону должен быть нечувствителен к регистру символов, а флаг `g` – что поиск должен быть глобальным, т. е. должны быть найдены все соответствия в строке. Флаг `m` выполняет поиск по шаблону в многострочном режиме. Если строковое выражение, в котором выполняется поиск, содержит символы перевода строк, то в этом режиме якорные символы `^` и `$`, помимо того, что они соответствуют началу и концу всего строкового выражения, также соответствуют началу и концу каждой текстовой строки. Например, шаблону `/java$/im` соответствует как слово «java», так и «Java\nis fun».

Эти флаги могут объединяться в любые комбинации. Например, чтобы выполнить поиск первого вхождения слова «java» (или «Java», «JAVA» и т. д.) без учета регистра символов, можно воспользоваться нечувствительным к регистру регулярным выражением `/\bjava\b/i`. А чтобы найти все вхождения этого слова в строке, можно добавить флаг `g`: `/\bjava\b/gi`.

В табл. 10.6 приводится перечень флагов регулярных выражений. Заметим, что флаг `g` более подробно рассматривается далее в этой главе вместе с методами классов `String` и `RegExp`, используемых для фактической реализации поиска.

Таблица 10.6. Флаги регулярных выражений

Символ	Значение
`i`	Выполняет поиск, нечувствительный к регистру.
`g`	Выполняет глобальный поиск, т. е. находит все соответствия, а не останавливается после первого из них.
`m`	Многострочный режим. `^` соответствует началу строки или началу всего строкового выражения, а `$` – концу строки или всего выражения.

10.2. Методы класса String для поиска по шаблону

До этого момента мы обсуждали грамматику создаваемых регулярных выражений, но не рассматривали, как эти регулярные выражения могут фактически использоваться в JavaScript-сценариях. В данном разделе мы обсудим методы объекта `String`, в которых регулярные выражения применяются для поиска по шаблону, а также для поиска с заменой. А затем продолжим разговор о поиске по шаблону с регулярными выражениями, рассмотрев объект `RegExp`, его методы и свойства. Обратите внимание, что последующее обсуждение – лишь обзор различных методов и свойств, относящихся к регулярным выражениям. Как обычно, полное описание можно найти в третьей части книги.

Строки поддерживают четыре метода, использующие регулярные выражения. Простейший из них – метод `search()`. Он принимает в качестве аргумента регулярное выражение и возвращает либо позицию первого символа найденной подстроки, либо -1, если соответствие не найдено. Например, следующий вызов вернет 4:

```
"JavaScript".search(/script/i);
```

Если аргумент метода `search()` не является регулярным выражением, он сначала преобразуется путем передачи конструктору `RegExp`. Метод `search()` не поддерживает глобальный поиск и игнорирует флаг `g` в своем аргументе.

Метод `replace()` выполняет операцию поиска с заменой. Он принимает в качестве первого аргумента регулярное выражение, а в качестве второго – строку замены.

Метод отыскивает в строке, для которой он вызван, соответствие указанному шаблону. Если регулярное выражение содержит флаг `g`, метод `replace()` заменяет все найденные совпадения строкой замены. В противном случае он заменяет только первое найденное совпадение. Если первый аргумент метода `replace()` является строкой, а не регулярным выражением, то метод выполняет буквальный поиск строки, а не преобразует его в регулярное выражение с помощью конструктора `RegExp()`, как это делает метод `search()`. В качестве примера мы можем воспользоваться методом `replace()` для единообразной расстановки прописных букв в слове «JavaScript» для всей строки текста:

```
// Независимо от регистра символов заменяем словом в нужном регистре
text.replace(/JavaScript/gi, "JavaScript");
```

Метод `replace()` представляет собой более мощное средство, чем можно было бы предположить по этому примеру. Напомню, что подвыражения в скобках, находящиеся внутри регулярного выражения, нумеруются слева направо, и что регулярное выражение запоминает текст, соответствующий каждому из подвыражений. Если в строке замены присутствует знак $ с цифрой, метод `replace()` заменяет эти два символа текстом, соответствующим указанному подвыражению. Это очень полезная возможность. Мы можем использовать ее, например, для замены прямых кавычек в строке типографскими кавычками, которые имитируются ASCII-символами:

```
// Цитата - это кавычка, за которой следует любое число символов, отличных от кавычек
// (их мы запоминаем), за этими символами следует еще одна кавычка.
var quote = /"([^"]*)"/g;
// Заменяем прямые кавычки типографскими и оставляем без изменений
// содержимое цитаты, хранящееся в $1.
text.replace(quote, "«$1»");
```

Метод `replace()` предоставляет и другие ценные возможности, о которых рассказывается в третьей части книги, в справке к методу `String.replace()`. Самое важное, что следует отметить, – второй аргумент `replace()` может быть функцией, динамически вычисляющей строку замены.

Метод `match()` – это наиболее общий из методов класса `String`, использующих регулярные выражения. Он принимает в качестве единственного аргумента регулярное выражение (или преобразует свой аргумент в регулярное выражение, передав его конструктору `RegExp()`) и возвращает массив, содержащий результаты поиска. Если в регулярном выражении установлен флаг `g`, метод возвращает массив всех соответствий, присутствующих в строке. Например:

```
"1 плюс 2 равно 3".match(/\d+/g) // вернет ["1", "2", "3"]
```

Если регулярное выражение не содержит флаг `g`, метод `match()` не выполняет глобальный поиск; он просто ищет первое совпадение. Однако `match()` возвращает массив, даже когда метод не выполняет глобальный поиск. В этом случае первый элемент массива – это найденная подстрока, а все оставшиеся элементы представляют собой подвыражения регулярного выражения. Поэтому если `match()` возвращает массив `a`, то `a[0]` будет содержать найденную строку целиком, `a[1]` – подстроку, соответствующую первому подвыражению, и т. д. Проводя параллель с методом `replace()`, можно сказать, что в `a[`*n*`]` заносится содержимое `$`*n*.

Например, взгляните на следующий программный код, выполняющий разбор URL-адреса:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Посетите мою домашнюю страницу http://www.example.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];  // Содержит "http://www.example.com/~david"
    var protocol = result[1]; // Содержит "http"
    var host = result[2];     // Содержит "www.example.com"
    var path = result[3];     // Содержит "~david"
}
```

Следует отметить, что для регулярного выражения, в котором не установлен флаг `g` глобального поиска, метод `match()` возвращает то же значение, что и метод `exec()` регулярного выражения: возвращаемый массив имеет свойства `index` и `input`, как описывается в обсуждении метода `exec()` ниже.

Последний из методов объекта `String`, в котором используются регулярные выражения, – `split()`. Этот метод разбивает строку, для которой он вызван, на массив подстрок, используя аргумент в качестве разделителя. Например:

```
"123,456,789".split(","); // Вернет ["123","456","789"]
```

Метод `split()` может также принимать в качестве аргумента регулярное выражение. Это делает метод более мощным. Например, можно указать разделитель, допускающий произвольное число пробельных символов с обеих сторон:

```
"1, 2, 3 , 4 ,5".split(/\s*,\s*/); // Вернет ["1","2","3","4","5"]
```

Метод `split()` имеет и другие возможности. Полное описание приведено в третьей части книги при описании метода `String.split()`.

10.3. Объект RegExp

Как было упомянуто в начале этой главы, регулярные выражения представлены в виде объектов `RegExp`. Помимо конструктора `RegExp()`, объекты `RegExp` поддерживают три метода и несколько свойств. Методы поиска и свойства класса `RegExp` описаны в следующих двух подразделах.

Конструктор `RegExp()` принимает один или два строковых аргумента и создает новый объект `RegExp`. Первый аргумент конструктора – это строка, содержащая тело регулярного выражения, т. е. текст, который должен находиться между символами слэша в литерале регулярного выражения. Обратите внимание, что в строковых литералах и регулярных выражениях для обозначения управляющих последовательностей используется символ \, поэтому, передавая конструктору `RegExp()` регулярное выражение в виде строкового литерала, необходимо заменить каждый символ \ парой символов \\. Второй аргумент `RegExp()` может отсутствовать. Если он указан, то определяет флаги регулярного выражения. Это должен быть один из символов `g`, `i`, `m` либо комбинация этих символов. Например:

```
// Находит все пятизначные числа в строке. Обратите внимание
// на использование в этом примере символов \\
var zipcode = new RegExp("\\d{5}", "g");
```

Конструктор `RegExp()` удобно использовать, когда регулярное выражение создается динамически и поэтому не может быть представлено с помощью синтаксиса литералов регулярных выражений. Например, чтобы найти строку, введенную пользователем, надо создать регулярное выражение во время выполнения с помощью `RegExp()`.

10.3.1. Свойства RegExp

Каждый объект `RegExp` имеет пять свойств. Свойство `source` – строка, доступная только для чтения, содержащая текст регулярного выражения. Свойство `global` – логическое значение, доступное только для чтения, определяющее наличие флага `g` в регулярном выражении. Свойство `ignoreCase` – это логическое значение, доступное только для чтения, определяющее наличие флага `i` в регулярном выражении. Свойство `multiline` – это логическое значение, доступное только для чтения, определяющее наличие флага `m` в регулярном выражении. И последнее свойство `lastIndex` – это целое число, доступное для чтения и записи. Для шаблонов с флагом `g` это свойство содержит номер позиции в строке, с которой должен быть начат следующий поиск. Как описано ниже, оно используется методами `exec()` и `test()`.

10.3.2. Методы RegExp

Объекты `RegExp` определяют два метода, выполняющие поиск по шаблону; они ведут себя аналогично методам класса `String`, описанным выше. Основной метод класса `RegExp`, используемый для поиска по шаблону, – `exec()`. Он похож на упоминавшийся метод `match()` класса `String`, за исключением того, что является методом класса `RegExp`, принимающим в качестве аргумента строку, а не методом класса `String`, принимающим аргумент `RegExp`. Метод `exec()` выполняет регулярное выражение для указанной строки, т. е. ищет совпадение в строке. Если совпадение не найдено, метод возвращает `null`. Однако если соответствие найдено, он возвращает такой же массив, как массив, возвращаемый методом `match()` для поиска без флага `g`. Нулевой элемент массива содержит строку, соответствующую регулярному выражению, а все последующие элементы – подстроки, соответствующие всем подвыражениям. Кроме того, свойство `index` содержит номер позиции символа, которым начинается соответствующий фрагмент, а свойство `input` ссылается на строку, в которой выполнялся поиск.

В отличие от `match()`, метод `exec()` возвращает массив, структура которого не зависит от наличия в регулярном выражении флага `g`. Напомню, что при передаче глобального регулярного выражения метод `match()` возвращает массив найденных соответствий. А `exec()` всегда возвращает одно соответствие, но предоставляет о нем полную информацию. Когда `exec()` вызывается для регулярного выражения, содержащего флаг `g`, метод устанавливает свойство `lastIndex` объекта регулярного выражения равным номеру позиции символа, следующего непосредственно за найденной подстрокой. Когда метод `exec()` вызывается для того же регулярного выражения второй раз, он начинает поиск с символа, позиция которого указана в свойстве `lastIndex`. Если `exec()` не находит соответствия, свойство `lastIndex` получает значение 0. (Вы также можете установить `lastIndex` в ноль в любой момент, что следует делать во всех тех случаях, когда поиск завершается до того, как будет найдено последнее соответствие в одной строке, и начинается поиск в другой строке с тем же объектом `RegExp`.) Это особое поведение позволяет вызывать `exec()` повторно для перебора всех соответствий регулярному выражению в строке. Например:

```
var pattern = /Java/g;
var text = "JavaScript - это более забавная штука, чем Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Найдено '" + result[0] + "'" +
          " в позиции " + result.index +
          "; следующий поиск начнется с " + pattern.lastIndex);
}
```

Еще один метод объекта `RegExp` – `test()`, который намного проще метода `exec()`. Он принимает строку и возвращает `true`, если строка соответствует регулярному выражению:

```
var pattern = /java/i;
pattern.test("JavaScript"); // Вернет true
```

Вызов `test()` эквивалентен вызову `exec()`, возвращающему `true`, если `exec()` возвращает не `null`. По этой причине метод `test()` ведет себя так же, как метод `exec()` при вызове для глобального регулярного выражения: он начинает искать указанную строку с позиции, заданной свойством `lastIndex`, и если находит соответствие, устанавливает свойство `lastIndex` равным номеру позиции символа, непосредственно следующего за найденным соответствием. Поэтому с помощью метода `test()` можно так же сформировать цикл обхода строки, как с помощью метода `exec()`.

Методы `search()`, `replace()` и `match()` класса `String` не используют свойство `lastIndex`, в отличие от методов `exec()` и `test()`. На самом деле методы класса `String` просто сбрасывают `lastIndex` в 0. Если метод `exec()` или `test()` использовать с шаблоном, в котором установлен флаг `g`, и выполнить поиск в нескольких строках, то мы должны либо найти все соответствия в каждой строке, чтобы свойство `lastIndex` автоматически сбросилось в ноль (это происходит, когда последний поиск оказывается неудачным), либо явно установить свойство `lastIndex` равным нулю. Если этого не сделать, поиск в новой строке может начаться с некоторой произвольной позиции, а не с начала. Если регулярное выражение не включает флаг `g`, то вам не придется беспокоиться об этом. Имейте также в виду, что в ECMAScript 5, когда интерпретатор встречает литерал регулярного выражения, он создает новый объект `RegExp`, со своим собственным свойством `lastIndex`, что снижает риск использования «левого» значения `lastIndex` по ошибке.

11

Подмножества и расширения JavaScript

До сих пор в книге описывалась официальная версия языка JavaScript, соответствующая стандартам ECMAScript 3 и ECMAScript 5. В этой главе, напротив, будет идти речь о подмножествах и надмножествах языка JavaScript. Подмножества языка были определены, по большей части, для обеспечения более высокого уровня безопасности: сценарий, использующий только безопасное подмножество языка, может использоваться без опаски, даже если он был получен из непроверенного источника, такого как рекламный сервер. В разделе 11.1 описываются некоторые из этих подмножеств.

Стандарт ECMAScript 3 был опубликован в 1999 году, и прошло десять лет, прежде чем стандарт был обновлен до версии ECMAScript 5, вышедшей в 2009 году. Брендан Эйх (Brendan Eich), создатель JavaScript, продолжал развивать язык на протяжении всех этих десяти лет (спецификация ECMAScript явно разрешает расширение языка) и совместно с проектом Mozilla выпустил версии JavaScript 1.5, 1.6, 1.7, 1.8 и 1.8.1 в Firefox 1.0, 1.5, 2, 3 и 3.5. Некоторые из расширенных особенностей JavaScript вошли в стандарт ECMAScript 5, но многие остаются нестандартизованными. Однако, как ожидается, некоторые из оставшихся нестандартных особенностей будут стандартизованы в будущем.

Эти расширения поддерживаются броузером Firefox, точнее, реализованным в нем интерпретатором Spidermonkey языка JavaScript. Созданный проектом Mozilla интерпретатор Rhino языка JavaScript, написанный на языке Java (раздел 12.1), также поддерживает большинство расширений. Однако, поскольку эти расширения языка не являются стандартными, они не особенно полезны для веб-разработчиков, которым требуется обеспечить совместимость своих веб-приложений со всеми броузерами. Тем не менее они описываются в этой главе, потому что они:

- чрезвычайно мощные;
- могут быть стандартизованы в будущем;
- могут использоваться при разработке расширений для Firefox;
- могут использоваться для разработки серверных сценариев на языке JavaScript, когда используется интерпретатор Spidermonkey или Rhino (раздел 12.1).

Вводный раздел был посвящен подмножествам языка, остальная же часть этой главы описывает расширения языка. Поскольку они нестандартизованы, они подаются в виде учебных материалов, с меньшим количеством строгих определений, чем особенности языка, описываемые в других главах книги.

11.1. Подмножества JavaScript

Большая часть подмножеств языка была определена с целью обеспечения безопасности при выполнении программного кода, полученного из непроверенных источников. Однако существует одно интересное подмножество языка, которое было создано по другим причинам. Мы рассмотрим его в первую очередь, а затем перейдем к подмножествам языка, цель которых заключается в обеспечении безопасности.

11.1.1. Подмножество The Good Parts

Небольшая книга Дугласа Крокфорда (Douglas Crockford) «JavaScript: The Good Parts» (O'Reilly) описывает подмножество JavaScript, включающее части языка, которые, по мнению автора, достойны использования. Цель этого подмножества – упростить язык, скрыть его недостатки и в конечном счете сделать программирование проще, а программы – лучше. Крокфорд так объясняет свои устремления:

> Большинство языков программирования имеют свои достоинства и недостатки. Я обнаружил, что могу писать более качественные программы, используя достоинства и избегая недостатков.

Подмножество Крокфорда не включает инструкции `with` и `continue`, а также функцию `eval()`. Оно позволяет определять функции только с помощью выражений определения и не содержит инструкции определения функций. Подмножество требует, чтобы тела циклов и условных инструкций заключались в фигурные скобки: оно не позволяет опускать скобки, даже если тело состоит из единственной инструкции. Оно требует, чтобы любая инструкция, не заканчивающаяся фигурной скобкой, завершалась точкой с запятой.

Подмножество не включает оператор точки, битовые операторы и операторы `++` и `--`. В нем также не допускается использовать операторы `==` и `!=` из-за выполняемых ими преобразований типов и требуется использовать вместо них операторы `===` и `!==`.

Поскольку в языке JavaScript отсутствует понятие области видимости блока, подмножество Крокфорда разрешает использовать инструкцию `var` только на верхнем уровне тела функции и требует, чтобы программист объявлял все локальные переменные функции с помощью единственной инструкции `var` – первой в теле функции. Подмножество не поощряет использование глобальных переменных, но это уже простое соглашение по программированию, а не фактическое ограничение языка.

Созданный Крокфордом веб-инструмент проверки качества программного кода (*http://jslint.com*) включает возможность проверки на соответствие требованиям подмножества Good Parts. Помимо проверок на отсутствие в программном коде недопустимых особенностей, инструмент JSLint также проверяет соблюдение общепринятых правил оформления, таких как корректное оформление отступов.

Крокфорд написал свою книгу до того, как в ECMAScript 5 был определен строгий режим, и теперь многие «недостатки» JavaScript, использование которых он осуждает в своей книге, могут быть запрещены за счет использования строгого режима. Теперь, с принятием стандарта ECMAScript 5, инструмент JSLint требует, чтобы программы включали директиву «use strict», если перед проверкой был включен параметр «The Good Parts».

11.1.2. Безопасные подмножества

Good Parts – это подмножество языка, созданное исходя из эстетических соображений и желания повысить производительность труда программиста. Существует также обширная категория подмножеств, созданных с целью повышения безопасности при выполнении программного кода JavaScript в безопасном окружении, или в «песочнице». Безопасные подмножества запрещают использование особенностей языка и библиотек, которые позволяют программному коду вырваться за пределы «песочницы» и влиять на глобальное окружение. Каждое подмножество снабжается статическим инструментом проверки, который анализирует программный код, чтобы убедиться, что он соответствует требованиям подмножества. Поскольку подмножества языка, которые могут пройти статическую проверку, обычно оказываются довольно ограниченными, некоторые системы организации безопасного окружения определяют большее, не так сильно ограничивающее, подмножество, добавляют этап трансформации программного кода, на котором выполняется проверка его соответствия более широкому подмножеству, и производят трансформацию программного кода для использования с более узким подмножеством языка. А кроме того, добавляют проверки времени выполнения в тех случаях, когда статический анализ программного кода не гарантирует полную безопасность.

Чтобы обеспечить приемлемый уровень безопасности при статической проверке, из языка JavaScript должны быть исключены следующие особенности:

- Ни в одном безопасном подмножестве не допускается использовать функцию `eval()` и конструктор `Function()`, потому что они позволяют выполнять произвольные строки программного кода, которые невозможно проверить статически.
- Ограничивается или полностью исключается возможность использовать ключевое слово `this`, потому что функции (в нестрогом режиме) с помощью `this` могут получить доступ к глобальному объекту. Предотвращение доступа к глобальному объекту является одной из основных целей любых систем организации безопасного окружения.
- Зачастую в безопасных подмножествах запрещается использовать инструкцию `with`, потому что она усложняет статическую проверку программного кода.
- В безопасных подмножествах не допускается использовать некоторые глобальные переменные. В клиентском JavaScript объект окна броузера дублирует глобальный объект, поэтому программному коду запрещается ссылаться на объект `window`. Аналогично клиентский объект `document` определяет методы, обеспечивающие полный контроль над содержимым страницы. Это слишком мощный инструмент, чтобы доверить его программному коду, не вызывающему доверия. Безопасные подмножества могут обеспечивать два разных подхода

к глобальным переменным, таким как `document`. Они могут полностью запрещать их использование и определять дополнительные функции, которые могут использоваться программным кодом, заключенным в безопасное окружение, для доступа к ограниченной части веб-страницы, выделенной для него. Другой подход заключается в том, что безопасное окружение, в котором выполняется программный код, может определять промежуточный объект `document`, реализующий только безопасную часть стандартного DOM API.

- В безопасных подмножествах не допускается использовать некоторые специальные свойства и методы, потому что они дают слишком широкие возможности потенциально небезопасному программному коду. В число таких свойств и методов обычно включаются свойства `caller` и `callee` объекта `arguments` (некоторые подмножества вообще запрещают использовать объект `arguments`), методы `call()` и `apply()` функций, а также свойства `constructor` и `prototype`. Кроме того, запрещается использовать нестандартные свойства, такие как `__proto__`. Некоторые подмножества идут по пути явного запрещения использования небезопасных свойств и глобальных переменных. Другие идут по пути разрешения доступа только к определенным, безопасным свойствам.
- Статический анализ с легкостью выявляет доступ к специальным свойствам, когда выражение доступа к свойству использует оператор точки. Но проверить присутствие обращений с помощью `[]` гораздо сложнее, потому что статический анализ не позволяет проверить все возможные варианты строковых выражений в квадратных скобках. По этой причине безопасные подмножества обычно запрещают использование квадратных скобок, если только выражение в них не является числовым или строковым литералом. Безопасные подмножества замещают операторы `[]` вызовами глобальных функций, выполняющих чтение и запись в свойства объектов – эти функции выполняют дополнительные проверки во время выполнения, чтобы убедиться, что программный код не пытается обратиться к запрещенным свойствам.

Некоторые из этих ограничений, такие как запрет на использование функции `eval()` и инструкции `with`, не слишком обременительны для программистов, потому что эти особенности обычно не используются при программировании на языке JavaScript. Другие, такие как ограничение на использование квадратных скобок для доступа к свойствам, являются достаточно тяжелыми, и здесь на помощь приходит механизм трансляции программного кода. Транслятор может автоматически преобразовать использование квадратных скобок, например, в вызовы функций, выполняющие дополнительные проверки во время выполнения. С помощью аналогичных трансформаций можно обезопасить использование ключевого слова `this`. Правда, в этом случае за безопасность приходится заплатить уменьшением скорости выполнения программного кода в безопасном окружении.

В настоящее время существует несколько безопасных подмножеств. И хотя полное их описание их выходит за рамки этой книги, тем не менее мы коротко познакомимся с некоторыми из наиболее известных:

ADsafe

Подмножество ADsafe (*http://adsafe.org*) было одним из первых предложенных безопасных подмножеств. Это подмножество было создано Дугласом Крокфордом (Douglas Crockford) (который также определил подмножество The Good Parts). Подмножество ADsafe опирается только на статическую проверку, которая выполняется с помощью инструмента JSLint (*http://jslint.org*). Оно запрещает доступ к большинству глобальных переменных и определяет переменную `ADSAFE`, которая предоставляет доступ к безопасным функциям, включая методы DOM специального назначения. Подмножество ADsafe не получило широкого распространения, но оно стало важным доказательством правильности самой концепции и послужило толчком к появлению других безопасных подмножеств.

dojox.secure

Подмножество `dojox.secure` (*http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/*) – это расширение для библиотеки Dojo (*http://dojotoolkit.org*), толчком к созданию которого послужило появление подмножества ADsafe. Подобно ADsafe, это подмножество основано на статической проверке использования ограниченного подмножества языка. В отличие от ADsafe, оно позволяет использовать стандартный DOM API. Кроме того, оно включает инструмент проверки, реализованный на языке JavaScript, благодаря чему имеется возможность динамической проверки программного кода перед его выполнением.

Caja

Подмножество Caja (*http://code.google.com/p/google-caja/*) – это подмножество, распространяемое с открытыми исходными текстами, созданное компанией Google. Подмножество Caja (по-испански «коробка») определяет два подмножества языка. Подмножество Cajita («маленькая коробка») – сильно ограниченное подмножество, подобное тому, которое используется подмножествами ADsafe и dojox.secure. Valija («кейс» или «чемодан») – намного более широкое подмножество языка, близкое к подмножеству строгого режима ECMAScript 5 (с исключением `eval()`). Само название Caja – это имя компилятора, трансформирующего веб-содержимое (HTML, CSS и программный код JavaScript) в безопасные модули, которые можно включать в веб-страницы, не опасаясь, что они будут оказывать влияние на страницу в целом или на другие модули.

Подмножество Caja – это часть OpenSocial API (*http://code.google.com/apis/opensocial/*); оно используется компанией Yahoo! на ее веб-сайтах. Например, содержимое, доступное на портале *http://my.yahoo.com*, организовано в модули Caja.

FBJS

Подмножество FBJS – это версия JavaScript, используемая на сайте Facebook (*http://facebook.com*) с целью дать пользователям возможность размещать непроверенное содержимое на страницах своих профилей. Подмножество FBJS обеспечивает безопасность за счет трансформации программного кода. Инструмент преобразования вставляет проверки, которые выполняются во время выполнения и предотвращают доступ к глобальному объекту с помощью ключевого слова `this`. Также он переименовывает все идентификаторы верхнего уровня, добавляя к ним префикс, определяемый модулем. Благодаря переименованию предотвращаются любые попытки изменить или прочитать глобальные переменные или переменные, принадлежащие другому модулю. Кроме того, благодаря добавлению префикса, все вызовы функции `eval()` преобразуются в вызовы несуществующей функции. Подмножество FBJS реализует собственное безопасное подмножество DOM API.

Microsoft Web Sandbox

Подмножество Microsoft Web Sandbox (*http://websandbox.livelabs.com/*) определяет довольно широкое подмножество языка JavaScript (плюс HTML и CSS) и обеспечивает безопасность за счет радикальной переработки программного кода, фактически реализуя безопасную виртуальную машину JavaScript поверх небезопасной.

11.2. Константы и контекстные переменные

Теперь оставим подмножества языка и перейдем к расширениям. В версии JavaScript 1.5 и выше появилась возможность использовать ключевое слово `const` для определения констант. Константы похожи на переменные, за исключением того, что попытки присваивания им значений игнорируются (они не вызывают ошибку), а попытка переопределить константу приводит к исключению:

```
const pi = 3.14; // Определить константу и дать ей значение.
pi = 4;          // Любые последующие операции присваивания игнорируются.
const pi = 4;    // Повторное объявление константы считается ошибкой.
var pi = 4;      // Это тоже ошибка.
```

Ключевое слово `const` действует подобно ключевому слову `var`: для него не существует области видимости блока и объявления константы поднимаются в начало вмещающего определения функции (раздел 3.10.1).

Отсутствие в языке JavaScript области видимости блока для переменных долгое время считали недостатком, поэтому в версии JavaScript 1.7 появилось ключевое слово `let`, решающее эту проблему. Ключевое слово `const` всегда было зарезервированным (но не используемым) словом в JavaScript, благодаря чему константы можно добавлять, не нарушая работоспособность существующего программного кода. Ключевое слово `let` не было зарезервировано, поэтому оно не распознается версиями интерпретаторов ниже 1.7.

Ключевое слово `let` имеет четыре формы использования:

- как инструкция объявления переменной, подобно инструкции `var`;
- в циклах `for` или `for/in`, как замена инструкции `var`;
- как инструкция блока, для объявления новых переменных и явного ограничения их области видимости; и
- для определения переменных, область видимости которых ограничивается единственным выражением.

В простейшем случае ключевое слово `let` используется как замена инструкции `var`. Переменные, объявленные с помощью инструкции `var`, определены в любой точке объемлющей функции. Переменные, объявленные с помощью ключевого слова `let`, определены только внутри ближайшего вмещающего блока (и, конечно же, внутри вложенных в него блоков). Если с помощью инструкции `let` объявить переменную, например, внутри тела цикла, она будет недоступна за пределами этого цикла:

```
function oddsums(n) {
    let total = 0, result=[];      // Определены в любой точке функции
    for(let x = 1; x <= n; x++) {  // переменная x определена только в цикле
```

```
        let odd = 2*x-1;          // переменная odd определена только в цикле
        total += odd;
        result.push(total);
    }
    // Попытка обратиться к переменной x или odd здесь
    // вызовет исключение ReferenceError
    return result;
}
oddsums(5);                       // Вернет [1,4,9,16,25]
```

Обратите внимание, что в этом примере инструкция `let` используется так же, как замена инструкции `var` в цикле `for`. Она создает переменную, которая будет доступна только в теле цикла и в выражениях проверки условия и увеличения цикла. Точно так же можно использовать инструкцию `let` в циклах `for/in` (и `for each`; раздел 11.4.1):

```
o = {x:1,y:2};
for(let p in o) console.log(p);         // Выведет x и y
for each(let v in o) console.log(v);    // Выведет 1 и 2
console.log(p)                          // ReferenceError: p не определена
```

Существует одно интересное отличие между случаями, когда `let` используется как инструкция объявления и когда `let` используется как инструмент инициализации переменной цикла. При использовании `let` в качестве инструкции объявления значение выражения инициализации вычисляется в области видимости переменной. Но в цикле `for` выражение инициализации вычисляется за пределами области видимости новой переменной. Это отличие становится важным, только когда имя новой переменной совпадает с именем уже существующей переменной:

```
let x = 1;
for(let x = x + 1; x < 5; x++)
    console.log(x);    // Выведет 2,3,4

{   // Начало блока, чтобы образовать область видимости новой переменной
    let x = x + 1;     // переменная x не определена, поэтому x+1 = NaN
    console.log(x);    // Выведет NaN
}
```

Переменные, объявленные с помощью инструкции `var`, определены в любой точке функции, где они объявлены, но они не инициализируются, пока инструкция `var` не будет выполнена фактически. То есть переменная существует (обращение к ней не вызывает исключение `ReferenceError`), но при попытке использовать переменную до инструкции `var` она будет иметь значение `undefined`. Переменные, объявляемые с помощью инструкции `let`, действуют аналогично: если попытаться использовать переменную до инструкции `let` (но внутри блока, где находится инструкция `let`), переменная будет доступна, но она будет иметь значение `undefined`.

Примечательно, что данная проблема отсутствует при использовании `let` для объявления переменной цикла – просто сам синтаксис не позволяет использовать переменную до ее инициализации. Существует еще один случай использования инструкции `let`, где отсутствует проблема использования переменной до ее инициализации. Блок, образуемый инструкцией `let` (в противоположность объявле-

нию переменной с помощью инструкции `let`, показанному выше), объединяет блок программного кода с объявлением переменных для этого блока и их инициализацией. В этом случае переменные и выражения их инициализации заключаются в круглые скобки, за которыми следует блок инструкций в фигурных скобках:

```
let x=1, y=2;
let (x=x+1,y=x+2) {   // Отметьте, что здесь выполняется сокрытие переменных
    console.log(x+y); // Выведет 5
};
console.log(x+y);     // Выведет 3
```

Важно запомнить, что выражения инициализации переменных `let`-блока не являются частью этого блока и интерпретируются в другой области видимости. В примере выше создается новая переменная `x`, и ей присваивается значение, на единицу больше, чем значение существующей переменной `x`.

Последний случай использования ключевого слова `let` является разновидностью `let`-блока, в котором вслед за списком переменных и выражений инициализации в круглых скобках следует единственное выражение, а не блок инструкций. Такая конструкция называется `let`-выражением, а пример выше можно было переписать, как показано ниже:

```
let x=1, y=2;
console.log(let (x=x+1,y=x+2) x+y); // Выведет 5
```

Некоторые формы использования ключевых слов `const` и `let` (не обязательно все четыре, описанные здесь) в будущем наверняка будут включены в стандарт ECMAScript.

Версии JavaScript

В этой главе при упоминании какой-то определенной версии JavaScript подразумевается версия языка, реализованная проектом Mozilla в интерпретаторах Spidermonkey и Rhino и в веб-броузере Firefox.

Некоторые расширения языка, представленные здесь, определяют новые ключевые слова (такие как `let`), и, чтобы избежать нарушения работоспособности существующего программного кода, использующего эти ключевые слова, JavaScript требует явно указывать версию, чтобы иметь возможность использовать расширения. Если вы пользуетесь автономным интерпретатором Spidermonkey или Rhino, версию языка можно указать в виде параметра командной строки или вызовом встроенной функции `version()`. (Она ожидает получить номер версии, умноженный на сто. Чтобы получить возможность использовать ключевое слово `let`, нужно выбрать версию JavaScript 1.7, т.е. передать функции число 170.) В Firefox указать номер версии можно в теге `script`:

```
<script type="application/JavaScript; version=1.8">
```

11.3. Присваивание с разложением

В версии Spidermonkey 1.7 реализована разновидность составных инструкций присваивания, известная как *присваивание с разложением*. (Вы могли встречать присваивание с разложением в языках программирования Python или Ruby.) При присваивании с разложением значение справа от знака «равно» является массивом или объектом («составным» значением), а слева указывается одно или более имен переменных с применением синтаксиса, имитирующего литерал массива или объекта.

Инструкция присваивания с разложением извлекает (разлагает на составляющие) одно или более значений из значения справа и сохраняет в переменных, указанных слева. Кроме того, как и обычный оператор присваивания, присваивание с разложением может использоваться для инициализации вновь объявляемых переменных в инструкциях `var` и `let`.

Присваивание с разложением является простым и мощным инструментом при работе с массивами, и его особенно удобно использовать при работе с функциями, возвращающими массивы значений. Однако при использовании с объектами и вложенными объектами эта операция становится сложной и запутанной. Примеры, демонстрирующие простоту и сложность, приводятся ниже.

Следующий пример демонстрирует простоту присваивания с разложением при использовании с массивами значений:

```
let [x,y] = [1,2];      // То же, что и let x=1, y=2
[x,y] = [x+1,y+1];      // То же, что и x = x + 1, y = y+1
[x,y] = [y,x];          // Обмен значений двух переменных
console.log([x,y]);     // Выведет [3,2]
```

Обратите внимание, как присваивание с разложением упрощает работу с функциями, возвращающими массивы значений:

```
// Преобразует координаты [x,y] в полярные координаты [r,theta]
function polar(x,y) {
    return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];
}
// Преобразует полярные координаты в декартовы координаты
function cartesian(r,theta) {
    return [r*Math.cos(theta), r*Math.sin(theta)];
}

let [r,theta] = polar(1.0, 1.0);    // r=Math.sqrt(2), theta=Math.PI/4
let [x,y] = cartesian(r,theta);     // x=1.0, y=1.0
```

При выполнении присваивания с разложением количество переменных слева не обязательно должно совпадать с количеством элементов массива справа. Лишние переменные слева получат значения `undefined`, а лишние значения справа будут просто игнорироваться. Список переменных слева может включать дополнительные запятые, чтобы пропустить определенные значения справа:

```
let [x,y] = [1];        // x = 1, y = undefined
[x,y] = [1,2,3];        // x = 1, y = 2
[,x,,y] = [1,2,3,4];    // x = 2, y = 4
```

В JavaScript отсутствует синтаксическая конструкция, которая позволила бы присвоить переменной все оставшиеся или неиспользованные значения (как массив). Так, во второй строке в примере выше, отсутствует возможность присвоить переменной `y` остаток массива `[2,3]`.

Значением инструкции присваивания с разложением является полная структура данных справа, а не отдельные значения, извлеченные из нее. То есть из операторов присваивания можно составлять «цепочки», как показано ниже:

```
let first, second, all;
all = [first,second] = [1,2,3,4]; // first=1, second=2, all=[1,2,3,4]
```

Присваивание с разложением можно даже использовать для извлечения значений из вложенных массивов. В этом случае левая сторона инструкции присваивания должна выглядеть как литерал вложенного массива:

```
let [one, [twoA, twoB]] = [1, [2,2.5], 3]; // one=1, twoA=2, twoB=2.5
```

Присваивание с разложением можно также выполнять, когда справа находится объект. В этом случае конструкция слева должна выглядеть как литерал объекта: список пар имен свойств и имен переменных, разделенных запятыми, заключенный в фигурные скобки. Имя слева от каждого двоеточия – это имя свойства, а имя справа от каждого двоеточия – это имя переменной. Каждое свойство, указанное слева, будет отыскиваться в объекте справа от оператора присваивания, и его значение (или `undefined`) будет присвоено соответствующей переменной. Эта разновидность присваивания с разложением может сбивать с толку, особенно если в качестве имен свойств и переменных используются одни и те же идентификаторы. Необходимо понимать, что в примере ниже `r`, `g` и `b` – это имена свойств, а `red`, `green` и `blue` – имена переменных:

```
let transparent = {r:0.0, g:0.0, b:0.0, a:1.0}; // Цвет в формате RGBA
let {r:red, g:green, b:blue} = transparent;     // red=0.0,green=0.0,blue=0.0
```

Следующий пример копирует глобальные функции из объекта `Math` в переменные с целью упростить программирование большого количества тригонометрических операций:

```
// То же, что и let sin=Math.sin, cos=Math.cos, tan=Math.tan
let {sin:sin, cos:cos, tan:tan} = Math;
```

Подобно тому как присваивание с разложением может использоваться при работе с вложенными массивами, эта операция может использоваться при работе с вложенными объектами. В действительности, эти два синтаксиса можно комбинировать для описания произвольных структур данных. Например:

```
// Вложенная структура данных: объект содержит массив объектов
let data = {
    name: "присваивание с разложением",
    type: "расширение",
    impl: [{engine: "spidermonkey", version: 1.7},
           {engine: "rhino", version: 1.7}]
};

// Использовать присваивание с разложением для извлечения
// четырех значений из структуры данных
let ({name:feature, impl: [{engine:impl1, version:v1},{engine:impl2}]} = data) {
    console.log(feature); // Выведет "присваивание с разложением"
    console.log(impl1);     // Выведет "spidermonkey"
    console.log(v1);        // Выведет 1.7
    console.log(impl2);     // Выведет "rhino"
}
```

Имейте в виду, что подобные вложенные инструкции присваивания с разложением могут превратить программный код в трудночитаемые дебри, вместо того чтобы упростить его. Однако есть одна интересная закономерность, которая поможет вам разобраться в самых сложных случаях. Представьте сначала обычное присваивание (с единственным значением). После того как присваивание будет выполнено, переменную слева от оператора присваивания можно взять и использовать как выражение, которое будет возвращать присвоенное значение. Мы говорили, что в инструкции присваивания с разложением слева указывается синтаксическая конструкция, напоминающая литерал массива или объекта. Но обратите внимание, что после выполнения присваивания с разложением программный код слева, который выглядит как литерал массива или объекта, действительно будет интерпретироваться как обычный литерал массива или объекта: все необходимые переменные будут определены, и вы сможете копировать текст слева от знака «равно» и использовать его в своей программе как массив или объект.

11.4. Итерации

Проектом Mozilla в расширение JavaScript были добавлены новые способы выполнения итераций, включая цикл `for each`**, а также итераторы и генераторы в стиле языка Python. Они детально описываются ниже.**

11.4.1. Цикл for/each

Цикл `for/each` **– это новая инструкция цикла, определяемая стандартом E4X. E4X (ECMAScript for XML) – это расширение языка, позволяющее употреблять в программах на языке JavaScript теги языка XML и предоставляющее функции для работы с данными в формате XML. Стандарт E4X реализован далеко не во всех веб-броузерах, но он поддерживается реализацией JavaScript проекта Mozilla, начиная с версии 1.6 (в Firefox 1.5). В этом разделе мы рассмотрим только цикл** `for/each` **и особенности его использования с обычными объектами, не имеющими отношения к XML. Остальные подробности о E4X приводятся в разделе 11.7.**

Цикл `for each` **напоминает цикл** `for/in`**. Однако вместо итераций по свойствам объекта он выполняет итерации по значениям свойств:**

```
let o = {one: 1, two: 2, three: 3}
for(let p in o) console.log(p);       // for/in: выведет 'one', 'two', 'three'
for each (let v in o) console.log(v); // for/each: выведет 1, 2, 3
```

При использовании с массивами цикл `for/each` **выполняет итерации по элементам (а не по индексам) массива. Обычно он перечисляет их в порядке следования числовых индексов, но в действительности такой порядок не определяется стандартом и не является обязательным:**

```
a = ['один', 'два', 'три'];
for(let p in a) console.log(p);       // Выведет индексы массива 0, 1, 2
for each (let v in a) console.log(v); // Выведет элементы 'один', 'два', 'три'
```

Обратите внимание, что область применения цикла `for/each` не ограничивается элементами массива – он может перечислять значения перечислимых свойств объекта, включая перечислимые методы, унаследованные объектом. По этой причине обычно не рекомендуется использовать цикл `for/each` для работы с объектами. Это особенно верно для программ, которые должны выполняться под управлением версий интерпретаторов JavaScript до ECMAScript 5, в которых невозможно сделать пользовательские свойства и методы неперечислимыми. (Смотрите аналогичное обсуждение цикла `for/in` в разделе 7.6.)

11.4.2. Итераторы

В версии JavaScript 1.7 цикл `for/in` был дополнен более универсальными возможностями. Цикл `for/in` в JavaScript 1.7 стал больше похож на цикл `for/in` в языке Python, он позволяет выполнять итерации по любым итерируемым объектам. Прежде чем обсуждать новые возможности, необходимо дать некоторые определения.

Итератором называется объект, который позволяет выполнять итерации по некоторой коллекции значений и хранит информацию о текущей «позиции» в коллекции. Итератор должен иметь метод `next()`. Каждый вызов метода `next()` должен возвращать следующее значение из коллекции. Например, функция `counter()`, представленная ниже, возвращает итератор, который, в свою очередь, возвращает последовательность увеличивающихся целых чисел при каждом вызове метода `next()`. Обратите внимание, что здесь для хранения текущей информации используется область видимости функции, образующая замыкание:

```
// Функция, возвращающая итератор;
function counter(start) {
    let nextValue = Math.round(start); // Частная переменная итератора
    return { next: function() { return nextValue++; }}; // Вернуть итератор
}

let serialNumberGenerator = counter(1000);
let sn1 = serialNumberGenerator.next(); // 1000
let sn2 = serialNumberGenerator.next(); // 1001
```

При работе с конечными коллекциями метод `next()` итератора возбуждают исключение `StopIteration`, когда в коллекции не остается значений для выполнения очередной итерации. `StopIteration` – это свойство глобального объекта в JavaScript 1.7. Его значением является обычный объект (без собственных свойств), зарезервированный специально для нужд завершения итераций. Обратите внимание, что `StopIteration` не является функцией-конструктором, таким как `TypeError()` или `RangeError()`. Ниже приводится пример метода `rangeIter()`, возвращающего итератор, который выполняет итерации по целым числам в заданном диапазоне:

```
// Функция, возвращающая итератор диапазона целых чисел
function rangeIter(first, last) {
    let nextValue = Math.ceil(first);
    return {
        next: function() {
            if (nextValue > last) throw StopIteration;
            return nextValue++;
        }
    };
}

// Пример неудобной реализации итераций с помощью итератора диапазона.
let r = rangeIter(1,5);             // Получить объект-итератор
while(true) {                       // Теперь использовать его в цикле
    try {
        console.log(r.next());      // Вызвать метод next() итератора
    }
    catch(e) {
        if (e == StopIteration) break; // Завершить цикл по StopIteration
        else throw e;
    }
}
```

Обратите внимание, насколько неудобно использовать объект-итератор в цикле из-за необходимости явно обрабатывать исключение `StopIteration`. Из-за этого неудобства итераторы редко используются на практике непосредственно. Чаще используются *итерируемые* объекты. Итерируемый объект представляет коллекцию значений, по которым можно выполнять итерации. Итерируемый объект должен определять метод с именем `__iterator__()` (с двумя символами подчеркивания в начале и в конце), возвращающий объект-итератор для коллекции.

В JavaScript 1.7 в цикл `for/in` была добавлена возможность работы с итерируемыми объектами. Если значение справа от ключевого слова `in` является итерируемым объектом, то цикл `for/in` автоматически вызовет его метод `__iterator__()`, чтобы получить объект-итератор. Затем он будет вызывать метод `next()` итератора, присваивать возвращаемое им значение переменной цикла и выполнять тело цикла. Цикл `for/in` сам обрабатывает исключение `StopIteration`, и оно никогда не передается программному коду, выполняемому в цикле. Пример ниже определяет функцию `range()`, возвращающую итерируемый объект (а не итератор), который представляет диапазон целых чисел. Обратите внимание, насколько проще выглядит цикл `for/in` при использовании итерируемого объекта диапазона по сравнению с циклом `while`, в котором используется итератор диапазона.

```
// Возвращает итерируемый объект, представляющий диапазон чисел
function range(min,max) {
    return {                        // Вернуть объект, представляющий диапазон.
        get min() { return min; },  // Границы диапазона не изменяются
        get max() { return max; },  // и хранятся в замыкании.
        includes: function(x) {     // Диапазоны могут проверять вхождение.
            return min <= x && x <= max;
        },
        toString: function() {      // Диапазоны имеют строковое представление.
            return "[" + min + "," + max + "]";
        },
        __iterator__: function() { // Возможно выполнять итерации по диапазону
            let val = Math.ceil(min); // Сохранить текущ. позицию в замыкании.
            return {                // Вернуть объект-итератор.
            next: function() {      // Вернуть следующее число в диапазоне.
                if (val > max)      // Если достигнут конец - прервать итерации
                    throw StopIteration;
                return val++;       // Иначе вернуть следующее число
            }                       // и увеличить позицию
            };
        }
    };
}

// Далее демонстрируется, как можно выполнять итерации по диапазону:
for(let i in range(1,10)) console.log(i); // Выведет числа от 1 до 10
```

Обратите внимание, что, несмотря на необходимость писать метод `__iterator__()` и возбуждать исключение `StopIteration` для создания итерируемых объектов и их итераторов, вам не придется (в обычной ситуации) вызывать метод `__iterator__()` и/или обрабатывать исключение `StopIteration` – все это сделает цикл `for/in`. Если по каким-то причинам потребуется явно получить объект-итератор итерируемого объекта, можно воспользоваться функцией `Iterator()`. (`Iterator()` – это глобальная функция, которая появилась в версии JavaScript 1.7.) Если передать этой функции итерируемый объект, она просто вернет результат вызова метода `__iterator__()`, что придаст дополнительную ясность программному коду. (Если передать функции `Iterator()` второй аргумент, она передаст его методу `__iterator__()`.)

Однако функция `Iterator()` имеет еще одно важное назначение. Если ей передать объект (или массив), не имеющий метода `__iterator__()`, она вернет собственную реализацию итерируемого итератора для объекта. Каждый вызов метода `next()` этого итератора будет возвращать массив с двумя значениями. В первом элементе массива будет возвращаться имя свойства объекта, а во втором – значение этого свойства. Поскольку этот объект является итерируемым итератором, его можно использовать в цикле `for/in` вместо прямого вызова метода `next()`, а это означает, что функцию `Iterator()` можно использовать совместно с операцией присваивания с разложением при выполнении итераций по свойствам и значениям объекта или массива:

```
for(let [k,v] in Iterator({a:1,b:2}))      // Итерации по ключам и значениям
    console.log(k + "=" + v);              // Выведет "a=1" и "b=2"
```

Итератор, возвращаемый функцией `Iterator()`, имеет еще две важные особенности. Во-первых, он игнорирует унаследованные свойства и выполняет итерации только по «собственным» свойствам, что чаще всего и требуется. Во-вторых, если передать функции `Iterator()` значение `true` во втором аргументе, возвращаемый итератор будет выполнять итерации только по именам свойств, без их значений. Обе эти особенности демонстрируются в следующем примере:

```
o = {x:1, y:2}                             // Объект с двумя свойствами
Object.prototype.z = 3;                    // Теперь все объекты унаследуют z
for(p in o) console.log(p);                // Выведет "x", "y" и "z"
for(p in Iterator(o, true)) console.log(p); // Выведет только "x" и "y"
```

11.4.3. Генераторы

Генераторы – это особенность JavaScript 1.7 (заимствованная из языка Python), основанная на использовании нового ключевого слова `yield`. Программный код, использующий данную особенность, должен явно указать номер версии 1.7, как описывалось в разделе 11.2. Ключевое слово `yield` используется в функциях и действует аналогично инструкции `return`, возвращая значение из функции. Разница между `yield` и `return` состоит в том, что функция, возвращающая значение с помощью ключевого слова `yield`, сохраняет информацию о своем состоянии, благодаря чему ее выполнение может быть возобновлено. Такая способность к возобновлению выполнения делает `yield` замечательным инструментом для создания итераторов. Генераторы – очень мощная особенность языка, но понять принцип их действия совсем не просто. Для начала познакомимся с некоторыми определениями.

Любая функция, использующая ключевое слово `yield` (даже если инструкция `yield` никогда не будет выполняться), является *функцией-генератором*. Функции-генераторы возвращают значения с помощью `yield`. Они могут использовать инструкцию `return` без значения, чтобы завершиться до того, как будет достигнут конец тела функции, но они не могут использовать `return` со значением. За исключением использования ключевого слова `yield` и ограничений на использование инструкции `return`, функции-генераторы ничем не отличаются от обычных функций: они объявляются с помощью ключевого слова `function`, оператор `typeof` возвращает для них строку «function» и как обычные функции они наследуют свойства и методы от `Function.prototype`. Однако поведение функции-генератора совершенно отличается от поведения обычной функции: при вызове, вместо того чтобы выполнить свое тело, функция-генератор возвращает объект *генератора*.

Генератор – это объект, представляющий текущее состояние функции-генератора. Он определяет метод `next()`, вызов которого возобновляет выполнение функции-генератора и позволяет продолжить ее выполнение, пока не будет встречена следующая инструкция `yield`. Когда это происходит, значение в инструкции `yield` в функции-генераторе становится возвращаемым значением метода `next()` генератора. Если функция-генератор завершает свою работу вызовом инструкции `return` или в результате достижения конца своего тела, метод `next()` генератора возбуждает исключение `StopIteration`.

Тот факт, что генераторы имеют метод `next()`, который может возбуждать исключение `StopIteration`, явственно говорит о том, что они являются итераторами.[1] В действительности они являются итерируемыми итераторами, т. е. они могут использоваться в циклах `for/in`. Следующий пример демонстрирует, насколько просто создавать функции-генераторы и выполнять итерации по значениям, которые они возвращают с помощью инструкции `yield`:

```
// Определение функции-генератора для выполнения итераций
// по целым числам в определенном диапазоне
function range(min, max) {
    for(let i = Math.ceil(min); i <= max; i++) yield i;
}

// Вызвать функцию-генератор, чтобы получить генератор, и выполнить итерации по нему.
for(let n in range(3,8)) console.log(n); // Выведет числа от 3 до 8.
```

---

[1] Генераторы иногда называют «итераторами-генераторами», чтобы отличить их от создающих их функций-генераторов. В этой главе вместо термина «итераторы-генераторы» мы будем использовать термин «генераторы». В других источниках вы можете встретить термин «генератор», обозначающий одновременно и функции-генераторы, и итераторы-генераторы.

Функции-генераторы могут никогда не завершаться. Каноническим примером использования генераторов является воспроизведение последовательности чисел Фибоначчи:

```
// Функция-генератор, которая воспроизводит последовательность чисел Фибоначчи
function fibonacci() {
    let x = 0, y = 1;
    while(true) {
        yield y;
         [x,y] = [y,x+y];
    }
}
// Вызвать функцию-генератор, чтобы получить генератор.
f = fibonacci();
// Использовать генератор как итератор, вывести первые 10 чисел Фибоначчи.
for(let i = 0; i < 10; i++) console.log(f.next());
```

Обратите внимание, что функция-генератор `fibonacci()` никогда не завершится. По этой причине создаваемый ею генератор никогда не возбудит исключение `StopIteration`. Поэтому, вместо того чтобы использовать его как итерируемый объект в цикле `for/in` и попасть в бесконечный цикл, мы используем его как итератор и явно вызываем его метод `next()` десять раз. После того как фрагмент выше будет выполнен, генератор `f` по-прежнему будет хранить информацию о состоянии функции-генератора. Если в программе не требуется далее хранить эту информацию, ее можно освободить вызовом метода `close()` объекта `f`:

```
f.close();
```

При вызове метода `close()` генератора производится завершение связанной с ним функции-генератора, как если бы она выполнила инструкцию `return` в той точке, где ее выполнение было приостановлено. Если это произошло в блоке `try`, автоматически будет выполнен блок `finally` перед тем, как `close()` вернет управление. Метод `close()` никогда не возвращает значение, но если блок `finally` возбудит исключение, оно продолжит свое распространение из вызова `close()`.

Генераторы часто бывает удобно использовать для последовательной обработки данных – элементов списка, строк текста, лексем в лексическом анализаторе и т. д. Генераторы можно объединять в цепочки, подобно конвейеру команд в Unix. Самое интересное в этом подходе заключается в том, что он следует принципу *отложенных вычислений*: значения «извлекаются» из генератора (или из конвейера генераторов) по мере необходимости, а не все сразу. Эту особенность демонстрирует пример 11.1.

Пример 11.1. Конвейер генераторов

```
// Генератор, возвращающий строки текста по одной.
// Обратите внимание, что здесь не используется метод s.split(), потому что
// он обрабатывает текст целиком, создает массив, тогда как нам необходимо
// реализовать отложенную обработку.
function eachline(s) {
    let p;
    while((p = s.indexOf('\n')) != -1) {
        yield s.substring(0,p);
        s = s.substring(p+1);
    }
    if (s.length > 0) yield s;
}

// Функция-генератор, возвращающая f(x) для каждого элемента x итерируемого объекта i
function map(i, f) {
    for(let x in i) yield f(x);
}

// Функция-генератор, возвращающая элементы i, для которых f(x) возвращает true
function select(i, f) {
    for(let x in i) {
        if (f(x)) yield x;
    }
}

// Обрабатываемый текст
let text = " #comment \n \n hello \nworld\n quit \n unreached \n";

// Сконструировать конвейер генераторов для обработки текста.
// Сначала разбить текст на строки
let lines = eachline(text);
// Затем удалить начальные и конечные пробелы в каждой строке
let trimmed = map(lines, function(line) { return line.trim(); });
// Наконец, игнорировать пустые строки и комментарии
let nonblank = select(trimmed, function(line) {
    return line.length > 0 && line[0] != "#"
});

// Теперь извлечь отфильтрованные строки из конвейера и обработать их,
// остановиться, если встретится строка "quit".
for (let line in nonblank) {
    if (line === "quit") break;
    console.log(line);
}
```

Обычно инициализация генераторов выполняется при их создании: аргументы, передаваемые функции-генератору, являются единственными значениями, которые принимают генераторы. Однако имеется возможность передать дополнительные данные уже работающему генератору. Каждый генератор имеет метод `send()`, который перезапускает генератор подобно методу `next()`. Разница лишь в том, что методу `send()` можно передать значение, которое станет значением, возвращаемым выражением `yield` в функции-генераторе. (В большинстве генераторов, которые не принимают дополнительных входных данных, ключевое слово `yield` выглядит как инструкция. Однако в действительности `yield` – это выражение, возвращающее значение.) Кроме методов `next()` и `send()` существует еще один способ перезапустить генератор – метод `throw()`. Если вызвать этот метод, выражение `yield` возбудит аргумент метода `throw()` как исключение, как показано в следующем примере:

```
// Функция-генератор, ведущая счет от заданного начального значения.
// Метод send() позволяет увеличить счетчик на определенное значение.
// Вызов throw("reset") сбрасывает счетчик в начальное значение.
// Это всего лишь пример - здесь метод throw() используется не самым лучшим образом.
function counter(initial) {
    let nextValue = initial;                 // Сохранить начальное значение
    while(true) {
        try {
            let increment = yield nextValue; // Вернуть значение и получить приращение
            if (increment)                  // Если передано приращение...
                nextValue += increment;     // ...использовать его.
            else nextValue++;               // Иначе увеличить на 1
        }
        catch (e) {                         // Если был вызван метод
            if (e==="reset")                // throw() генератора
                nextValue = initial;
            else throw e;
        }
    }
}

let c = counter(10);           // Создать генератор с начальным значением 10
console.log(c.next());         // Выведет 10
console.log(c.send(2));        // Выведет 12
console.log(c.throw("reset")); // Выведет 10
```

11.4.4. Генераторы массивов

Еще одна особенность, заимствованная в JavaScript 1.7 из языка Python, – *генераторы массивов*. Это механизм инициализации элементов массива на основе элементов другого массива или итерируемого объекта. Синтаксис генераторов массивов основан на математической форме записи элементов множества, т. е. выражения и инструкции находятся совсем не там, где привыкли их видеть программисты на языке JavaScript. Тем не менее привыкание к необычному синтаксису происходит достаточно быстро, а мощь генераторов массивов просто неоценима.

Ниже приводится пример генератора массивов, использующего созданную выше функцию `range()` для инициализации массива, содержащего квадраты четных чисел, меньшие 100:

```
let evensquares = [x*x for (x in range(0,10)) if (x % 2 === 0)]
```

Эта строка примерно эквивалентна следующим пяти строкам:

```
let evensquares = [];
for(x in range(0,10)) {
    if (x % 2 === 0)
        evensquares.push(x*x);
}
```

В общем случае синтаксис генераторов массивов имеет следующий вид:

`[` *выражение* `for (` *переменная* `in` *объект* `) if (` *условное выражение* `) ]`

Обратите внимание на три основные части в квадратных скобках:

- Цикл `for/in` или `for/each` без тела. Эта часть генератора массивов включает *переменную* (или несколько переменных при использовании присваивания с разложением) слева от ключевого слова `in` и *объект* (который может быть генератором, итерируемым объектом или массивом) справа от ключевого слова `in`. Несмотря на отсутствие тела цикла, эта часть генератора массивов выполняет итерации и присваивает последовательные значения, определяемые переменной. Обратите внимание, что перед именем переменной не допускается указывать ключевое слово `var` или `let` – генераторы массивов неявно используют ключевое слово `let`, а используемая переменная недоступна за пределами квадратных скобок и не затирает существующую переменную с тем же именем.

- После итерируемого объекта может присутствовать ключевое слово `if` и *условное выражение*. Если оно присутствует, условное выражение используется для фильтрации значений, по которым выполняются итерации. Условное выражение вычисляется после получения каждого значения, воспроизводимого циклом `for`. Если результатом выражения является `false`, это значение пропускается и в массив ничего не добавляется. Ключевое слово `if` можно не указывать – если оно отсутствует, генератор массива действует так, как если бы в нем присутствовала конструкция `if (true)`.
- *Выражение*, стоящее перед ключевым словом `for`, представляет собой эквивалент тела цикла. После того как значение, возвращаемое итератором, будет присвоено *переменной* и пройдет проверку *условным выражением*, будет вычислено значение этого выражения, и полученный результат будет добавлен в создаваемый массив.

Ниже приводятся несколько более конкретных примеров, которые помогут лучше понять синтаксис:

```
data = [2,3,4, -5];                    // Массив чисел
squares = [x*x for each (x in data)]; // Квадраты всех чисел: [4,9,16,25]
// Извлечь квадратные корни из всех неотрицательных элементов
roots = [Math.sqrt(x) for each (x in data) if (x >= 0)]

// Создать массив с именами свойств объекта
o = {a:1, b:2, f: function(){}}
let allkeys = [p for (p in o)]
let ownkeys = [p for (p in o) if (o.hasOwnProperty(p))]
let notfuncs = [k for ([k,v] in Iterator(o)) if (typeof v !== "function")]
```

11.4.5. Выражения-генераторы

В JavaScript 1.8[1] можно заменить квадратные скобки в генераторах массивов круглыми скобками и получить выражения-генераторы. ***Выражение-генератор*** похоже на генератор массивов (синтаксис в круглых скобках в точности соответствует синтаксису в квадратных скобках), но его значением является объект генератора, а не массив. Преимущество выражений-генераторов перед генераторами массивов в том, что они используют прием отложенных вычислений – вычисления выполняются по мере необходимости, а не все сразу – и позволяют обрабатывать даже бесконечные последовательности. Недостаток генераторов состоит в том, что они обеспечивают только последовательный доступ к своим элементам. То есть, в отличие от массивов, генераторы не позволяют обращаться к элементам по индексам: чтобы получить *n*-е значение, придется выполнить *n-1* итераций.

Ранее в этой главе мы реализовали функцию `map()`:

```
// Функция-генератор, возвращающая f(x) для каждого элемента x итерируемого объекта i
```

---

[1] На момент написания этих строк выражения-генераторы не поддерживались в Rhino.

```
function map(i, f) {
    for(let x in i) yield f(x);
}
```

Выражения-генераторы позволяют избежать необходимости создавать или использовать такую функцию `map()`. Чтобы получить новый генератор `h`, возвращающий `f(x)` для каждого значения `x`, возвращаемого генератором `g`, достаточно использовать такой программный код:

```
let h = (f(x) for (x in g));
```

Используя генератор `eachline()` из примера 11.1, можно реализовать отсечение пробельных символов, а также фильтрацию комментариев и пустых строк, как показано ниже:

```
let lines = eachline(text);
let trimmed = (l.trim() for (l in lines));
let nonblank = (l for (l in trimmed) if (l.length > 0 && l[0]!='#'));
```

11.5. Краткая форма записи функций

В JavaScript 1.8[1] появилась возможность краткой записи простых функций (называется «лексическим замыканием»). Если функция вычисляет единственное выражение и возвращает его значение, ключевое слово `return` и фигурные скобки, окружающие тело функции, можно опустить и просто поместить выражение сразу после списка аргументов. Например:

```
let succ = function(x) x+1, yes = function() true, no = function() false;
```

Это просто и удобно: функции, определяемые таким способом, ведут себя как обычные функции, в определении которых присутствуют фигурные скобки и ключевое слово `return`. Этот сокращенный синтаксис удобно использовать, в частности, при передаче функций другим функциям. Например:

```
// Отсортировать массив в обратном порядке
data.sort(function(a,b) b-a);

// Определение функции, которая возвращает сумму квадратов элементов массива
let sumOfSquares = function(data)
    Array.reduce(Array.map(data, function(x) x*x), function(x,y) x+y);
```

11.6. Множественные блоки catch

В JavaScript 1.5 инструкция `try/catch` была добавлена возможность использовать несколько блоков `catch`. Чтобы использовать эту возможность, необходимо, чтобы за именем параметра блока `catch` следовало ключевое слово `if` и условное выражение:

```
try {
    // здесь могут возбуждаться исключения нескольких типов
throw 1;
```

---

[1] На момент написания этих строк данная особенность не поддерживалась в Rhino.

```
}
catch(e if e instanceof ReferenceError) {
    // Здесь обрабатывается исключение обращения к неопределенному имени
}
catch(e if e === "quit") {
    // Обработка исключения, получаемое в результате возбуждения сроки "quit"
}
catch(e if typeof e === "string") {
    // Здесь обрабатываются все остальные строковые исключения
}
catch(e) {
    // Здесь обрабатываются любые другие исключения
}
finally {
    // Блок finally действует как обычно
}
```

Когда возникает какое-либо исключение, по очереди проверяются все блоки `catch`. Исключение присваивается именованному параметру блока `catch` и вычисляется условное выражение. Если оно возвращает `true`, выполняется тело этого блока `catch`, а все остальные блоки `catch` пропускаются. Если блок `catch` не имеет условного выражения, он ведет себя как блок с условным выражением `if true` и выполняется всегда, если перед ним не был встречен блок `catch`, удовлетворяющий условию. Если условное выражение присутствует во всех блоках `catch` и ни в одном из них условное выражение не вернуло `true`, исключение продолжит распространение как необработанное. Обратите внимание, что, поскольку условное выражение уже находится в круглых скобках блока `catch`, его не требуется еще раз заключать в скобки, как в обычных инструкциях `if`.

11.7. E4X: ECMAScript for XML

Расширение ECMAScript for XML, более известное как E4X, – это стандартное расширение[1] JavaScript, определяющее ряд мощных особенностей для обработки XML-документов. Расширение E4X поддерживается интерпретаторами Spidermonkey 1.5 и Rhino 1.6. Из-за того что оно не получило широкой поддержки у производителей броузеров, расширение E4X, вероятно, лучше относить к серверным технологиям, основанным на интерпретаторах Spidermonkey или Rhino.

Расширение E4X представляет XML-документ (или элементы и атрибуты XML-документа) как объект `XML`, и представляет фрагменты XML (несколько элементов XML, не имеющих общего родителя) в виде родственного объекта `XMLList`. В этом разделе мы познакомимся с несколькими способами создания и обработки объектов `XML`. Объекты `XML` – это совершенно новый тип объектов, для работы с которыми в E4X предусмотрен (как мы увидим) специальный синтаксис. Как известно, для всех стандартных объектов JavaScript, не являющихся функциями, оператор `typeof` возвращает строку «object». Объекты `XML`, подобно функциям, отличаются от обычных объектов JavaScript, и для них оператор `typeof` возвращает строку

---

[1] Расширение E4X определяется стандартом ECMA-357. Официальную спецификацию можно найти по адресу *http://www.ecma-international.org/publications/standards/Ecma-357.htm*.

«xml». Важно понимать, что объекты XML никак не связаны с объектами DOM (Document Object Model – объектная модель документа), которые используются в клиентском JavaScript (глава 15). Стандарт E4X определяет дополнительные средства для преобразования XML-документов и элементов между представлениями E4X и DOM, но броузер Firefox не реализует их. Это еще одна причина, почему расширение E4X лучше относить к серверным технологиям.

Этот раздел представляет собой краткое учебное руководство по расширению E4X и не должен рассматриваться как полноценное его описание. В частности, объекты XML и XMLList имеют множество методов, которые вообще не будут упоминаться здесь. Их описание также отсутствует в справочном разделе. Тем из вас, у кого появится желание использовать расширение E4X, за более полной информацией необходимо обращаться к официальной спецификации.

Расширение E4X определяет совсем немного новых синтаксических конструкций. Самая заметная часть нового синтаксиса заключается в возможности использования разметки XML непосредственно в программном коде JavaScript и включения в него литералов XML, как показано ниже:

```
// Создать объект XML
var pt =
    <periodictable>
        <element id="1"><name>Водород</name></element>
        <element id="2"><name>Гелий</name></element>
        <element id="3"><name>Литий</name></element>
    </periodictable>;

// Добавить новый элемент в таблицу
pt.element += <element id="4"><name>Бериллий</name></element>;
```

Синтаксис литералов XML в расширении E4X в качестве экранирующих символов использует угловые скобки, что позволяет помещать в разметку XML произвольные выражения на языке JavaScript. Ниже демонстрируется еще один способ создания точно такого же элемента XML, как в примере выше:

```
pt = <periodictable></periodictable>;               // Создать пустую таблицу
var elements = ["Водород", "Гелий", "Литий"];       // Добавить элементы
// Создать теги XML, используя содержимое массива
for(var n = 0; n < elements.length; n++) {
    pt.element += <element id={n+1}><name>{elements[n]}</name></element>;
}
```

В дополнение к синтаксису литералов можно также извлекать данные из строк с разметкой XML. Следующий пример добавляет в периодическую таблицу еще один элемент:

```
pt.element += new XML('<element id="5"><name>Бор</name></element>');
```

При работе с фрагментами XML вместо конструктора XML() используется конструктор XMLList():

```
pt.element += new XMLList('<element id="6"><name>Углерод</name></element>' +
                          '<element id="7"><name>Азот</name></element>');
```

После создания XML-документа для доступа к его содержимому можно использовать интуитивно понятный синтаксис E4X:

```
var elements = pt.element;   // Вернет список всех тегов <element>
var names = pt.element.name; // Список всех тегов <name>
var n = names[0];            // "Водород": содержимое тега <name> с номером 0.
```

Кроме того, расширение E4X добавляет новый синтаксис для работы с объектами `XML`. Оператор `..` – это оператор доступа к вложенным элементам. Его можно использовать вместо привычного оператора `.` доступа к членам:

```
// Другой способ получить список всех тегов <name>
var names2 = pt..name;
```

Расширение E4X позволяет использовать даже оператор шаблона:

```
// Получить все вложенные теги <element>.
// Это еще один способ получить список всех тегов <name>.
var names3 = pt.element.*;
```

Расширение E4X отличает имена атрибутов от имен тегов с помощью символа `@` (этот синтаксис заимствован из языка XPath). Например, значение атрибута можно запросить, как показано ниже:

```
// Получить атомное число гелия
var atomicNumber = pt.element[1].@id;
```

Оператор шаблона для имен атрибутов имеет вид `@*`:

```
// Список свех атрибутов всех тегов <element>
var atomicNums = pt.element.@*;
```

Расширение E4X включает даже мощные и удивительно выразительные синтаксические конструкции для фильтрации списков с помощью произвольных выражений-предикатов:

```
// Отфильтровать список всех элементов так, чтобы
// он включал только элементы с атрибутом id < 3
var lightElements = pt.element.(@id < 3);

// Отфильтровать список всех тегов <element> так, чтобы он включал только те,
// имена которых начинаются с символа "Б". Затем создать список тегов <name>
// из оставшихся тегов <element>.
var bElementNames = pt.element.(name.charAt(0) == 'Б').name;
```

Цикл `for/each`, с которым мы познакомились выше в этой главе (раздел 11.4.1), в расширении E4X дополнен возможностью итераций по спискам тегов и атрибутов XML. Напомню, что цикл `for/each` похож на цикл `for/in`, за исключением того, что вместо итераций по свойствам объекта он выполняет итерации по значениям свойств:

```
// Вывести названия всех элементов периодической таблицы
for each (var e in pt.element) {
    console.log(e.name);
}

// Вывести атомные числа элементов
for each (var n in pt.element.@*) console.log(n);
```

В расширении E4X выражения могут присутствовать слева от оператора присваивания. Это позволяет изменять существующие и добавлять новые теги и атрибуты:

```
// Изменить тег <element> для Водорода - добавить в него новый атрибут
// и новый дочерний элемент, чтобы он выглядел так:
//
// <element id="1" symbol="B">
//     <name>Водород</name>
//     <weight>1.00794</weight>
// </element>
//
pt.element[0].@symbol = "B";
pt.element[0].weight = 1.00794;
```

Так же легко можно удалять теги и атрибуты, используя стандартный оператор `delete`:

```
delete pt.element[0].@symbol; // удалить атрибут
delete pt..weight;            // удалить все теги <weight>
```

Расширение E4X реализовано так, что позволяет выполнять большинство типичных операций с документами XML с помощью привычного синтаксиса языка. В E4X также имеются методы, которые можно вызывать относительно объектов `XML`. Например, метод `insertChildBefore()`:

```
pt.insertChildBefore(pt.element[1],
                     <element id="1"><name>Дейтерий</name></element>);
```

Расширение E4X полностью поддерживает пространства имен, а также включает синтаксические конструкции и функции для работы с пространствами имен XML:

```
// Объявить пространство имен по умолчанию с помощью инструкции
// "default xml namespace" statement:
default xml namespace = "http://www.w3.org/1999/xhtml";

// Следующий документ xhtml содержит несколько тегов svg:
d = <html>
        <body>
    Маленький красный квадрат:
    <svg xmlns="http://www.w3.org/2000/svg" width="10" height="10">
        <rect x="0" y="0" width="10" height="10" fill="red"/>
    </svg>
        </body>
    </html>

// Элемент body с uri его пространства имен и локальным именем
var tagname = d.body.name();
var bodyns = tagname.uri;
var localname = tagname.localName;

// Выборка элементов <svg> выполняется сложнее, потому что они не принадлежат
// пространству имен по умолчанию. Поэтому для выборки svg сначала нужно создать
// объект Namespace и затем использовать его
// оператор :: добавляет пространство имен в имя тега
var svg = new Namespace('http://www.w3.org/2000/svg');
var color = d..svg::rect.@fill          // "red"
```

12

Серверный JavaScript

В предыдущих главах подробно рассматривался базовый язык JavaScript, и вскоре мы перейдем ко второй части книги, в которой рассказывается о том, как JavaScript встраивается в веб-броузеры, и описывается обширнейший API клиентского JavaScript. JavaScript – это язык программирования для Веб, и большая часть программного кода на языке JavaScript написана для выполнения в веб-броузерах. Однако JavaScript – это быстрый и универсальный язык с широкими возможностями, и нет никаких причин, по которым JavaScript не мог бы использоваться для решения других задач программирования. Поэтому, прежде чем перейти к знакомству с клиентским JavaScript, мы коротко рассмотрим две другие реализации JavaScript. *Rhino* – это интерпретатор JavaScript, написанный на языке Java, что обеспечивает программам на языке JavaScript доступ ко всем библиотекам языка Java. Интерпретатор Rhino рассматривается в разделе 12.1. *Node* – версия интерпретатора V8 JavaScript, созданного компанией Google, включающая низкоуровневые интерфейсные библиотеки доступа к POSIX (Unix) API – к файлам, процессам, потокам, сокетам и так далее – с особым упором на асинхронные операции ввода/вывода, сетевые взаимодействия и работу с протоколом HTTP. Интерпретатор Node рассматривается в разделе 12.2.

Название главы говорит, что она посвящена «серверному» JavaScript, а обычно для создания серверов и для управления ими используются интерпретаторы Node и Rhino. Но под словом «серверный» можно также понимать «все, что за пределами веб-броузера». Программы, выполняемые под управлением Rhino, способны создавать графические интерфейсы пользователя, используя фреймворк Swing для языка Java. А интерпретатор Node может выполнять программы на языке JavaScript, способные манипулировать файлами подобно тому, как это делают сценарии командной оболочки.

Цель этой короткой главы состоит в том, чтобы осветить некоторые направления за пределами веб-броузеров, где может использоваться язык JavaScript. Здесь не предпринимается попытка в полном объеме охватить интерпретатор Rhino или Node, а обсуждаемые здесь прикладные интерфейсы не описываются в справочном разделе. Очевидно, что в одной главе невозможно сколько-нибудь полно опи-

сать платформу Java или POSIX API, поэтому раздел об интерпретаторе Rhino предполагает, что читатели имеют некоторое знакомство с Java, а раздел об интерпретаторе Node предполагает знакомство с низкоуровневыми прикладными интерфейсами Unix.

12.1. Управление Java с помощью Rhino

Rhino – это интерпретатор JavaScript, написанный на языке Java, цель которого – упростить возможность создания программ на языке JavaScript, которые могли бы использовать мощь платформы Java. Интерпретатор Rhino автоматически выполняет преобразование простых типов JavaScript в простые типы Java и наоборот, благодаря чему сценарии на языке JavaScript могут читать и изменять свойства и вызывать методы объектов на языке Java.

Rhino определяет несколько важных глобальных функций, не являющихся частью базового языка JavaScript:

```
// Специфические глобальные функции, определяемые интерпретатором:
// Введите help() в строке приглашения rhino, чтобы получить дополнительную информацию
print(x);              // Глобальная функция вывода в консоль
version(170);          // Требует от Rhino задействовать особенности версии 1.7
load(filename,...);    // Загружает и выполняет один или более файлов
                       // с программами на языке JavaScript
readFile(file);        // Читает текстовый файл и возвращает его содержимое в виде строки
readUrl(url);          // Читает текстовое содержимое по адресу URL и возвращает
                       // в виде строки
spawn(f);              // Вызывает f() или загружает и выполняет файл f
                       // в новом потоке выполнения
runCommand(cmd,        // Запускает системную команду с нулем или более
        [args...]);    // аргументов командной строки
quit()                 // Завершает работу Rhino
```

Как получить Rhino

Интерпретатор Rhino – свободное программное обеспечение, разработанное проектом Mozilla. Вы можете загрузить копию по адресу *http://www.mozilla.org/rhino/*. Версия Rhino 1.7r2 реализует ECMAScript 3 и ряд расширений языка, описанных в главе 11. Rhino – достаточно зрелый программный продукт, и новые его версии появляются не так часто. На момент написания этих строк в репозитории с исходными текстами была доступна предварительная версия 1.7r3, включающая частичную реализацию стандарта ECMAScript 5.

Rhino распространяется в виде JAR-архива. Запускается он командой, как показано ниже:

```
java -jar rhino1_7R2/js.jar program.js
```

Если опустить параметр *program.js*, интерпретатор Rhino запустится в интерактивном режиме, который позволит вам опробовать простенькие программы и однострочные примеры.

Обратите внимание на функцию `print()`: мы будем использовать ее в этом разделе вместо `console.log()`. Интерпретатор Rhino представляет пакеты и классы Java как объекты JavaScript:

```
// Глобальная переменная Packages определяет корень иерархии пакетов Java
Packages.any.package.name // Любой пакет из CLASSPATH
java.lang               // Глобальная переменная java - краткая ссылка на Packages.java
javax.swing              // А javax - краткая ссылка на Packages.javax

// Классы: доступны как свойства пакетов
var System = java.lang.System;
var JFrame = javax.swing.JFrame;
```

Поскольку пакеты и классы представлены как объекты JavaScript, их можно присваивать переменным, чтобы давать им более короткие имена. Но при желании их можно импортировать более формальным способом:

```
var ArrayList = java.util.ArrayList;   // Создать краткое имя для класса
importClass(java.util.HashMap);        // Аналог: var HashMap=java.util.HashMap

// Импорт пакета (отложенный) с помощью importPackage().
// Не следует импортировать java.lang: слишком много конфликтов имен
// с глобальными переменными JavaScript.
importPackage(java.util);
importPackage(java.net);

// Другой прием: передать произвольное количество классов и пакетов функции
// JavaImporter() и использовать возвращаемый объект с инструкцией with
var guipkgs = JavaImporter(java.awt, java.awt.event, Packages.javax.swing);
with (guipkgs) {
    /* Здесь определены такие классы, как Font, ActionListener и JFrame */
}
```

С помощью ключевого слова `new` можно создавать экземпляры классов языка Java, как если бы это были классы JavaScript:

```
// Объекты: создание из классов Java с помощью ключевого слова new
var f = new java.io.File("/tmp/test"); // Этот объект используется ниже
var out = new java.io.FileWriter(f);
```

Интерпретатор Rhino позволяет использовать JavaScript-оператор `instanceof` для работы с объектами и классами на языке Java:

```
f instanceof java.io.File          // => true
out instanceof java.io.Reader      // => false: объект Writer, а не Reader
out instanceof java.io.Closeable   // => true: Writer реализует Closeable
```

Как видно выше в примерах создания объектов экземпляров, интерпретатор Rhino позволяет передавать значения конструкторам Java и присваивать возвращаемые ими значения переменным JavaScript. (Обратите внимание на неявное преобразование типов, выполняемое интерпретатором Rhino в этом примере: JavaScript-строка «/tmp/test» автоматически преобразуется в значение типа *java.lang.String*.) Методы Java очень похожи на конструкторы Java, и Rhino позволяет программам на языке JavaScript вызывать методы на языке Java:

```
// Статические методы на языке Java действуют подобно функциям JavaScript
java.lang.System.getProperty("java.version")  // Вернет версию Java
```

```
var isDigit = java.lang.Character.isDigit;   // Присвоит статич. метод переменной
isDigit("٢")                                 // => true: Арабская цифра 2

// Вызвать методы экземпляра объектов f и out на языке Java, созданных выше
out.write("Hello World\n");
out.close();
var len = f.length();
```

Кроме того, Rhino позволяет получать и изменять значения статических полей Java-классов и полей экземпляров Java-объектов из программы на языке JavaScript. В классах на языке Java часто не определяют общедоступные поля, отдавая предпочтение методам доступа. Если в Java-классе определены методы доступа, Rhino обеспечивает доступ к ним, как к свойствам объекта на языке JavaScript:

```
// Прочитать значение статического поля Java-класса
var stdout = java.lang.System.out;

// Rhino отображает методы доступа в отдельные свойства JavaScript
f.name                        // => "/tmp/test": вызовет f.getName()
f.directory                   // => false: вызовет f.isDirectory()
```

В языке Java имеется возможность создавать перегруженные версии методов, имеющие одинаковые имена, но разные сигнатуры. Обычно интерпретатор Rhino способен определить, какую версию метода следует вызвать, опираясь на типы аргументов, которые передаются программой на языке JavaScript. Однако иногда бывает необходимо явно идентифицировать метод по имени и сигнатуре:

```
// Предположим, что Java-объект o имеет метод f, который принимает целое
// или вещественное число. В JavaScript необходимо будет явно указать сигнатуру:
o['f(int)'](3);               // Вызвать метод, принимающий целое число
o['f(float)'](Math.PI);       // Вызвать метод, принимающий вещественное число
```

Для итераций по методам, полям и свойствам Java-классов можно использовать цикл `for/in`:

```
importClass(java.lang.System);
for(var m in System) print(m); // Выведет статические члены java.lang.System
for(m in f) print(m);          // Выведет члены экземпляра java.io.File

// Обратите внимание, что таким способом нельзя перечислить классы в пакете
for (c in java.lang) print(c); // Этот прием не сработает
```

Rhino позволяет программам на языке JavaScript получать и изменять значения элементов Java-массивов, как если бы они были JavaScript-массивами. Конечно, Java-массивы отличаются от JavaScript-массивов: они имеют фиксированную длину, их элементы имеют определенный тип, и они не имеют JavaScript-методов, таких как `slice()`. В JavaScript не существует синтаксических конструкций, которые могли бы использоваться интерпретатором Rhino для создания Java-массивов в программах на языке JavaScript, поэтому для этой цели необходимо использовать класс *java.lang.reflect.Array*:

```
// Создать массив из 10 строк и массив из 128 байтов
var words = java.lang.reflect.Array.newInstance(java.lang.String, 10);
var bytes = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE, 128);
```

```
// После создания с массивами можно работать как с JavaScript-массивами:
for(var i = 0; i < bytes.length; i++) bytes[i] = i;
```

Программирование на языке Java часто связано с реализацией интерфейсов. Чаще всего с этой необходимостью приходится сталкиваться при разработке графических интерфейсов, когда каждый обработчик события должен реализовать интерфейс приемника событий. Следующие примеры демонстрируют, как это сделать:

```
// Интерфейсы: Реализация интерфейсов выглядит следующим образом:
var handler = new java.awt.event.FocusListener({
    focusGained: function(e) { print("got focus"); },
    focusLost: function(e) { print("lost focus"); }
});

// Аналогично выполняется расширение абстрактных классов
var handler = new java.awt.event.WindowAdapter({
    windowClosing: function(e) { java.lang.System.exit(0); }
});

// Когда интерфейс определяет единственный метод, можно использовать простую функцию
button.addActionListener(function(e) { print("button clicked"); });

// Если все методы интерфейса или абстрактного класса имеют одну и ту же сигнатуру,
// в качестве реализации можно использовать единственную функцию,
// а Rhino будет передавать ей имя метода в последнем аргументе
frame.addWindowListener(function(e, name) {
    if (name === "windowClosing") java.lang.System.exit(0);
});

// Если необходимо определить объект, реализующий несколько интерфейсов,
// можно использовать класс JavaAdapter:
var o = new JavaAdapter(java.awt.event.ActionListener, java.lang.Runnable, {
    run: function() {},              // Реализует интерфейс Runnable
    actionPerformed: function(e) {}  // Реализует интерфейс ActionListener
});
```

Когда Java-метод возбуждает исключение, интерпретатор Rhino продолжает его распространение как JavaScript-исключения. Получить оригинальный Java-объект *java.lang.Exception* можно через свойство `javaException` JavaScript-объекта `Error`:

```
try {
    java.lang.System.getProperty(null); // null - недопустимый аргумент
}
catch(e) {                          // e - JavaScript-исключение
    print(e.javaException); // это обертка для java.lang.NullPointerException
}
```

Здесь необходимо сделать последнее замечание по поводу преобразования типов в Rhino. Интерпретатор Rhino автоматически преобразует простые числа, логические значения и `null`. Java-тип `char` интерпретируется в языке JavaScript как число, так как в языке JavaScript отсутствует символьный тип. JavaScript-строки автоматически преобразуются в Java-строки, но (и это может быть камнем преткновения) Java-строки остаются объектами *java.lang.String* и не преобразуются

обратно в JavaScript-строки. Взгляните на следующую строку из примера, приводившегося ранее:

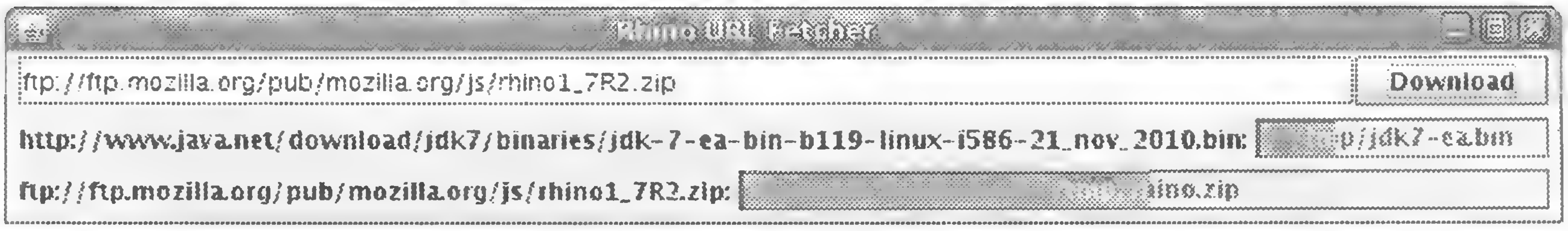
```
var version = java.lang.System.getProperty("java.version");
```

После выполнения этой инструкции переменная `version` **будет хранить объект** ***java.lang.String*****. Он обычно ведет себя как JavaScript-строка, но существуют важные отличия. Во-первых, Java-строка вместо свойства** `length` **имеет метод** `length()`**. Во-вторых, оператор** `typeof` **возвращает тип «object» для Java-строк. Java-строку нельзя преобразовать в JavaScript-строку вызовом метода** `toString()`**, потому что все Java-объекты имеют собственные методы** `toString()`**, возвращающие экземпляры** ***java.lang.String*****. Чтобы преобразовать Java-значение в строку, его нужно передать JavaScript-функции** `String()`**:**

```
var version = String(java.lang.System.getProperty("java.version"));
```

12.1.1. Пример использования Rhino

В примере 12.1 приводится простое приложение для интерпретатора Rhino, демонстрирующее большую часть возможностей и приемов, описанных выше. Пример использует пакет ***javax.swing*** **со средствами построения графических интерфейсов, пакет** ***java.net*** **с инструментами организации сетевых взаимодействий, пакет** ***java.io*** **потокового ввода/вывода и инструменты языка Java многопоточного выполнения для реализации простого приложения менеджера загрузки, которое загружает файлы по адресам URL и отображает ход выполнения загрузки. На рис. 12.1 показано, как выглядит окно приложения в процессе загрузки двух файлов.**



ftp://ftp.mozilla.org/pub/mozilla.org/js/rhino1_7R2.zip

http://www.java.net/download/jdk7/binaries/jdk-7-ea-bin-b119-linux-i586-21_nov_2010.bin:

ftp://ftp.mozilla.org/pub/mozilla.org/js/rhino1_7R2.zip:

Рис. 12.1. Графический интерфейс, созданный с помощью Rhino

Пример 12.1. Приложение менеджера загрузки для Rhino

```
/*
 * Приложение менеджера загрузки с простым графическим интерфейсом,
 * построенным средствами языка Java
 */
// Импортировать графические компоненты из библиотеки Swing
// и несколько других классов
importPackage(javax.swing);
importClass(javax.swing.border.EmptyBorder);
importClass(java.awt.event.ActionListener);
importClass(java.net.URL);
importClass(java.io.FileOutputStream);
importClass(java.lang.Thread);

// Создать графические элементы управления
var frame = new JFrame("Rhino URL Fetcher"); // Окно приложения
var urlfield = new JTextField(30);          // Поле ввода URL
var button = new JButton("Download");       // Кнопка запуска загрузки
var filechooser = new JFileChooser();       // Диалог выбора файла
var row = Box.createHorizontalBox();        // Контейнер для поля и кнопки
var col = Box.createVerticalBox();          // Для строки и индикатора хода
                                            // выполнения операции
var padding = new EmptyBorder(3,3,3,3);     // Отступы для строк

// Объединить все компоненты и отобразить графический интерфейс
row.add(urlfield);                          // Поместить поле ввода в строку
row.add(button);                            // Поместить кнопку в строку
col.add(row);                               // Поместить строку в колонку
frame.add(col);                             // Поместить колонку во фрейм
row.setBorder(padding);                     // Добавить отступы вокруг строки
frame.pack();                               // Определить минимальный размер
frame.visible = true;                       // Вывести окно

// Эта функция вызывается, когда в окне что-то происходит.
frame.addWindowListener(function(e, name) {
    // Если пользователь закрыл окно, завершить приложение.
    if (name === "windowClosing")           // Rhino добавляет аргумент name
        java.lang.System.exit(0);
});

// Эта функция вызывается, когда пользователь щелкает на кнопке
button.addActionListener(function() {
    try {
        // Создать обьект java.net.URL для представления URL источника.
        // (Автоматически будет проверена корректность ввода пользователя)
        var url = new URL(urlfield.text);
        // Предложить пользователю выбрать файл для сохранения содержимого URL
        var response = filechooser.showSaveDialog(frame);
        // Завершить, если пользователь щелкнул на кнопке Cancel
        if (response != JFileChooser.APPROVE_OPTION) return;
        // Иначе получить обьект java.io.File, представляющий файл назначения
        var file = filechooser.getSelectedFile();
        // Запустить новый поток выполнения для загрузки url
        new java.lang.Thread(function() { download(url,file); }).start();
    }
    catch(e) {
        // Вывести диалог, если что-то пошло не так
        JOptionPane.showMessageDialog(frame, e.message, "Exception",
                                      JOptionPane.ERROR_MESSAGE);
    }
});

// Использует java.net.URL и др. для загрузки содержимого URL и использует java.io.File
// и др. для сохранения этого содержимого в файле. Отображает ход выполнения загрузки
// в компоненте JProgressBar. Эта функция вызывается в новом потоке выполнения.
function download(url, file) {
    try {
        // Каждый раз, когда запускается загрузка очередного файла,
        // необходимо добавить в окно новую строку для отображения URL,
        // имени файла и индикатора хода выполнения операции
        var row = Box.createHorizontalBox(); // Создать строку
```

```
            row.setBorder(padding);                   // Добавить отступы
            var label = url.toString() + ": ";        // Отобразить URL
            row.add(new JLabel(label));               // в компоненте JLabel
            var bar = new JProgressBar(0, 100);       // Создать полосу индикатора
            bar.stringPainted = true;                 // Отобразить имя файла
            bar.string = file.toString();             // в полосе индикатора
            row.add(bar);                             // Добавить индикатор в строку
            col.add(row);                             // Добавить строку в колонку
            frame.pack();                             // Изменить размер окна

            // Здесь еще не известен размер загружаемого файла, поэтому изначально
            // индикатор просто воспроизводит анимационный эффект
            bar.indeterminate = true;

            // Установить соединение с сервером и получить размер загружаемого
            // файла, если это возможно
            var conn = url.openConnection();          // Получить java.net.URLConnection
            conn.connect();                           // Подключиться и ждать заголовки
            var len = conn.contentLength;             // Проверить, получена ли длина файла
            if (len) {                                // Если длина известна, тогда
                bar.maximum = len;                    // настроить индикатор на вывод
                bar.indeterminate = false;            // процента выполнения задания
            }

            // Получить потоки ввода и вывода
            var input = conn.inputStream;             // Прочитать байты с сервера
            var output = new FileOutputStream(file); // Записать в файл

            // Создать входной буфер в виде массива размером 4 Кбайта
            var buffer = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
                                                             4096);
            var num;
            while((num=input.read(buffer)) != -1) {  // Читать до признака EOF
                output.write(buffer, 0, num);         // Записать байты в файл
                bar.value += num;                     // Обновить индикатор
            }
            output.close();                           // Закрыть потоки по завершении
            input.close();
        }
        catch(e) { // Если что-то пошло не так, вывести ошибку в индикаторе
            if (bar) {
                bar.indeterminate = false;    // Остановить анимацию
                bar.string = e.toString();    // Заменить имя файла сообщением
            }
        }
    }
```

12.2. Асинхронный ввод/вывод в интерпретаторе Node

Node – это быстрый интерпретатор JavaScript, написанный на языке C++, включающий средства доступа к низкоуровневым интерфейсам Unix для работы с процессами, файлами, сетевыми сокетами и так далее, а также к клиентским и серверным интерфейсам реализации протокола HTTP. За исключением нескольких

синхронных методов, имеющих специальные имена, все остальные инструменты интерпретатора Node доступа к интерфейсам Unix являются асинхронными, и по умолчанию программы, выполняемые под управлением Node, никогда не блокируются, что обеспечивает им хорошую масштабируемость и позволяет эффективно справляться с высокой нагрузкой. Поскольку прикладные программные интерфейсы являются асинхронными, интерпретатор Node опирается на использование обработчиков событий, которые часто реализуются с использованием вложенных функций и замыканий.[1]

Этот раздел освещает некоторые наиболее важные инструменты и события, имеющиеся в составе Node, но это описание ни в коем случае нельзя считать полным. Полное описание Node можно найти в электронной документации по адресу *http://nodejs.org/api/*.

Как получить Node

Node – это свободное программное обеспечение, которое можно загрузить по адресу *http://nodejs.org*. На момент написания этих строк интерпретатор все еще активно разрабатывался и скомпилированные дистрибутивы не были доступны, однако вы можете собрать собственную копию интерпретатора из исходных текстов. Примеры в этом разделе опробовались под управлением версии Node 0.4. Прикладной интерфейс интерпретатора еще не зафиксирован, однако основные функции, демонстрируемые в этом разделе, едва ли сильно изменятся в будущем.

Интерпретатор Node построен на основе механизма V8 JavaScript, разработанного компанией Google. Версия Node 0.4 использует версию V8 3.1, которая реализует все особенности ECMAScript 5, за исключением строгого режима.

После загрузки, компиляции и установки Node вы сможете запускать программы, написанные для этого интерпретатора, как показано ниже:

```
node program.js
```

Знакомство с интерпретатором Rhino мы начали с функций `print()` и `load()`. Интерпретатор Node имеет похожие инструменты, но с другими именами:

```
// Подобно броузерам, для вывода отладочной информации Node определяет
// функцию console.log().
console.log("Hello Node"); // Выведет отладочную информацию в консоль

// Вместо load() в нем используется функция require().
// Она загружает и выполняет (только один) указанный модуль и возвращает объект,
// содержащий имена, экспортируемые модулем.
var fs = require("fs");      // Загрузит модуль "fs" и вернет объект с его API
```

---

[1] Клиентский JavaScript также является асинхронным и опирается на использование событий, поэтому вам будет проще понять примеры, которые приводятся в этом разделе, после того как вы прочтете вторую часть книги и познакомитесь с особенностями создания клиентских программ на языке JavaScript.

Интерпретатор Node реализует в глобальном объекте все стандартные конструкторы, свойства и функции, предусмотренные стандартом ECMAScript 5. Однако в дополнение к этому он также поддерживает клиентские функции для работы с таймером: `setTimeout()`**,** `setInterval()`**,** `clearTimeout()` **и** `clearInterval()`**:**

```
// Вывести приветствие через одну секунду.
setTimeout(function() { console.log("Привет, Мир!"); }, 1000);
```

Эти глобальные клиентские функции рассматриваются в разделе 14.1. Реализация Node совместима с реализациями интерпретаторов в веб-броузерах.

Интерпретатор Node также определяет и другие глобальные компоненты в пространстве имен `process`**. Ниже перечислены некоторые из свойств этого объекта:**

```
process.version  // Строка с версией Node
process.argv     // Аргументы командной строки в виде массива, argv[0] = "node"
process.env      // Переменные окружения в виде объекта.
                 // например: process.env.PATH
process.pid      // Числовой идентификатор процесса
process.getuid() // Возвращает числовой идентификатор пользователя
process.cwd()    // Возвращает текущий рабочий каталог
process.chdir()  // Выполняет переход в другой каталог
process.exit()   // Завершает программу (после запуска всех обработчиков)
```

Поскольку функции и методы, реализуемые интерпретатором Node, являются асинхронными, они не блокируют выполнение программы в ожидании завершения операций. Неблокирующий метод не может вернуть результат выполнения асинхронной операции. Если в программе потребуется получить результат или просто определить, когда завершится операция, необходимо определить функцию, которую интерпретатор Node сможет вызвать, когда результат будет доступен или когда операция завершится (или возникнет ошибка). В некоторых случаях (например, в вызове `setTimeout()` **выше) достаточно просто передать методу функцию в виде аргумента, и Node вызовет ее в соответствующий момент времени. В других случаях можно воспользоваться инфраструктурой событий интерпретатора Node. Объекты, реализованные в интерпретаторе Node, которые генерируют события (их часто называют *источниками* (*emitter*) событий), определяют метод** `on()` **для регистрации обработчиков. Они принимают в первом аргументе тип события (строку) и функцию-обработчик во втором аргументе. Для различных типов событий функциям-обработчикам передаются различные аргументы, поэтому вам может потребоваться обратиться к документации по API, чтобы точно узнать, как писать свои обработчики:**

```
emitter.on(name, f)             // Регистрирует f для обработки события name,
                                // генерируемого объектом emitter
emitter.addListener(name, f)    // То же самое: addListener() - синоним для on()
emitter.once(name, f)           // Для обработчиков однократного срабатывания,
                                // затем f автоматически удаляется
emitter.listeners(name)         // Возвращает массив функций-обработчиков
emitter.removeListener(name, f) // Удаляет обработчик f
emitter.removeAllListeners(name) // Удаляет все обработчики события name
```

Объект `process`**, представленный выше, является источником событий. Ниже приводится пример обработчиков некоторых его событий:**

```
// Событие "exit" отправляется перед завершением работы Node.
process.on("exit", function() { console.log("Goodbye"); });

// Необработанные исключения генерируют события, если имеется хотя бы один
// зарегистрированный обработчик. В противном случае исключение
// заставляет интерпретатор Node вывести сообщение и завершить работу.
process.on("uncaughtException", function(e) { console.log(Exception, e); });

// Сигналы POSIX, такие как SIGINT, SIGHUP и SIGTERM, также генерируют события
process.on("SIGINT", function() { console.log("Ignored Ctrl-C"); });
```

Поскольку интерпретатор Node позволяет выполнять высокопроизводительные операции ввода/вывода, его прикладной интерфейс к потокам ввода/вывода является одним из наиболее часто используемых в программах. Потоки, открытые для чтения, генерируют события, когда появляются данные, готовые для чтения. В следующем примере предполагается, что `s` – это поток, открытый для чтения, созданный где-то в другом месте программы. Ниже будет показано, как создавать объекты потоков для файлов и сетевых сокетов:

```
// Входной поток s:
s.on("data", f);     // При появлении данных передать их функции f() в аргументе
s.on("end", f);      // событие "end" возникает по достижении конца файла,
                     // когда данные больше не могут поступить
s.on("error", f);    // Если что-то не так, передаст исключение функции f()
s.readable           // => true, если поток по-прежнему открыт для чтения
s.pause();           // Приостановит отправку событий "data".
                     // Например, чтобы уменьшить скорость выгрузки
s.resume();          // Возобновит отправку событий "data"

// Определяет кодировку, если обработчику события "data" данные должны
// передаваться в виде строк
s.setEncoding(enc); // Как декодировать байты: "utf8", "ascii" или "base64"
```

Потоки, открытые для записи, не так тесно связаны с событиями, как потоки, открытые для чтения. Метод `write()` таких потоков используется для отправки данных, а метод `end()` – для закрытия потока, когда все данные будут записаны. Метод `write()` никогда не блокируется. Если интерпретатор Node окажется не в состоянии записать данные немедленно и во внутреннем буфере потока не окажется сводного места, метод `write()` вернет `false`. Чтобы узнать, когда интерпретатор вытолкнет буфер и данные фактически будут записаны, можно зарегистрировать обработчик события «drain»:

```
// Выходной поток s:
s.write(buffer);          // Запись двоичных данных
s.write(string, encoding) // Запись строковых данных.
                          // по умолчанию encoding = "utf-8"
s.end()                   // Закроет поток.
s.end(buffer);            // Запишет последнюю порцию данных и закроет поток.
s.end(str, encoding)      // Запишет последнюю строку и закроет поток
s.writeable;              // true, если поток по-прежнему открыт для записи
s.on("drain", f)          // f() будет вызываться при опустошении внутр. буфера
```

Как видно из примеров выше, потоки ввода/вывода, реализованные в интерпретаторе Node, могут работать и с двоичными, и с текстовыми данными. Текст передается с использованием простых строк JavaScript. Байты обрабатываются с помощью специфического для Node типа данных `Buffer`. Буферы в интерпретаторе Node являются объектами, подобными массивам, с фиксированной длиной, элементами которых могут быть только числа в диапазоне от 0 до 255. Программы, выполняющиеся под управлением Node, часто интерпретируют буферы как непрозрачные блоки данных, читая их из одного потока и записывая в другой. Тем не менее байты в буфере доступны как обычные элементы массива, а сами буферы имеют методы, позволяющие копировать байты из одного буфера в другой, получать срезы буфера, записывать строки в буфер с использованием заданной кодировки и декодировать буфер или его часть обратно в строку:

```
var bytes = new Buffer(256);         // Создать новый буфер на 256 байт
for(var i = 0; i < bytes.length; i++) // Цикл по индексам
    bytes[i] = i;                    // Установить каждый элемент в буфере
var end = bytes.slice(240, 256);     // Получить срез буфера
end[0]                               // => 240: end[0] = bytes[240]
end[0] = 0;                          // Изменить элемент среза
bytes[240]                           // => 0: буфер тоже изменится
var more = new Buffer(8);            // Создать новый отдельный буфер
end.copy(more, 0, 8, 16);            // Скопировать элементы 8-15 из end[] в more[]
more[0]                              // => 248

// Буферы также позволяют выполнять преобразования двоичных данных в строки
// и обратно. Допустимые кодировки: "utf8", "ascii" и "base64".
// По умолчанию используется "utf8".
var buf = new Buffer("2πr", "utf8"); // Закодировать текст в байты, UTF-8
buf.length                           // => 3 символа занимают 4 байта
buf.toString()                       // => "2πr": обратно в текст
buf = new Buffer(10);                // Новый буфер фиксированной длины
var len = buf.write("πr²", 4);       // Записать текст, начиная с 4-го байта
buf.toString("utf8",4, 4+len)        // => "πr²": декодировать диапазон байтов
```

Инструменты интерпретатора Node для работы с файлами и файловой системой находятся в модуле «fs»:

```
var fs = require("fs"); // Загрузит инструменты для работы с файловой системой
```

Для большинства своих методов этот модуль предоставляет синхронные версии. Любой метод, имя которого оканчивается на «Sync», является блокирующим методом, возвращающим значение и возбуждающим исключение. Методы для работы с файловой системой, имена которых не оканчиваются на «Sync», являются неблокирующими – они возвращают результаты или ошибки посредством передаваемых им функций обратного вызова. Следующий фрагмент демонстрирует, как реализуется чтение текстового файла с использованием блокирующего метода и как реализуется чтение двоичного файла с использованием неблокирующего метода:

```
// Синхронное чтение файла. Следует передать имя кодировки,
// чтобы в результате получить текст, а не двоичные байты.
var text = fs.readFileSync("config.json", "utf8");

// Асинхронное чтение двоичного файла. Следует передать функцию, чтобы получить данные
fs.readFile("image.png", function(err, buffer) {
    if (err) throw err;  // Если что-то пошло не так
    process(buffer);     // Содержимое файла в параметре buffer
});
```

Для записи в файл существуют аналогичные функции `writeFile()` **и** `writeFileSync()`**:**

```
fs.writeFile("config.json", JSON.stringify(userprefs));
```

Функции, представленные выше, интерпретируют содержимое файла как единственную строку или объект `Buffer`**. Кроме того, для чтения и записи файлов интерпретатор Node определяет также API потоков ввода/вывода. Функция ниже копирует содержимое одного файла в другой:**

```
// Копирование файлов с применением API потоков ввода/вывода.
// Чтобы определить момент окончания копирования,
// ей нужно передать функцию обратного вызова
function fileCopy(filename1, filename2, done) {
    var input = fs.createReadStream(filename1);       // Входной поток
    var output = fs.createWriteStream(filename2);     // Выходной поток

    input.on("data", function(d) { output.write(d); }); // Копировать
    input.on("error", function(err) { throw err; });    // Сообщить об ошибке
    input.on("end", function() {          // По исчерпании входных данных
        output.end();                     // закрыть выходной поток
        if (done) done();                 // И известить вызвавшую программу
    });
}
```

Модуль «fs» включает также несколько методов, возвращающих список содержимого каталогов, атрибуты файлов и т. д. Следующая ниже программа для интерпретатора Node использует синхронные методы для получения списка содержимого каталога, а также для определения размеров файлов и времени последнего их изменения:

```
#! /usr/local/bin/node
var fs = require("fs"), path = require("path");     // Загрузить модули
var dir = process.cwd();                            // Текущий каталог
if (process.argv.length > 2) dir = process.argv[2]; // Или из команд. строки
var files = fs.readdirSync(dir);                    // Прочитать содер. кат-га
process.stdout.write("Name\tSize\tDate\n");         // Вывести заголовок
files.forEach(function(filename) {                  // Для каждого файла
    var fullname = path.join(dir,filename);         // Объед. имя и каталог
    var stats = fs.statSync(fullname);              // Получить атрибуты файла
    if (stats.isDirectory()) filename += "/";       // Пометить подкаталоги
    process.stdout.write(filename + "\t" +          // Вывести имя файла
                         stats.size + "\t" +        // размер файла
                         stats.mtime + "\n");       // и время посл. изменения
});
```

Обратите внимание на комментарий `#!` **в первой строке, в примере выше. Это специальный комментарий, используемый в Unix, чтобы объявить сценарий, следующий далее, исполняемым, определив файл интерпретатора, который должен его выполнить. Интерпретатор Node игнорирует подобные строки комментариев, когда они находятся в первых строках файлов.**

Модуль «net» определяет API для организации взаимодействий по протоколу TCP. (Для выполнения сетевых взаимодействий на основе дейтаграмм можно использовать модуль «dgram».) Ниже приводится пример очень простого сетевого TCP-сервера, реализованного на основе особенностей Node:

```
// Простой эхо-сервер, реализованный на основе особенностей Node:
// он ожидает соединений на порту с номером 2000 и отправляет обратно клиенту
// все данные, которые получит от него.
var net = require('net');
var server = net.createServer();
server.listen(2000, function() { console.log("Прослушивается порт 2000"); });
server.on("connection", function(stream) {
    console.log("Принято соединение от", stream.remoteAddress);
    stream.on("data", function(data) { stream.write(data); });
    stream.on("end", function(data) { console.log("Соединение закрыто"); });
});
```

В дополнение к базовому модулю «net» в интерпретаторе Node имеется встроенная поддержка протокола HTTP в виде модуля «http». Особенности его использования демонстрируют примеры, следующие ниже.

12.2.1. Пример использования Node: HTTP-сервер

В примере 12.2 приводится реализация простого HTTP-сервера, основанная на особенностях интерпретатора Node. Она обслуживает файлы в текущем каталоге и дополнительно реализует два адреса URL специального назначения, которые обслуживаются особым образом. В этой реализации используется модуль «http», входящий в состав интерпретатора Node, и применяются API доступа к файлам и потокам ввода/вывода, демонстрировавшиеся выше. В примере 18.17, в главе 18, демонстрируется аналогичный специализированный HTTP-сервер.

Пример 12.2. HTTP-сервер, основанный на особенностях Node

```
// Простой NodeJS HTTP-сервер, обслуживающий файлы в текущем каталоге
// и реализующий два специальных адреса URL для нужд тестирования.
// Подключение к серверу выполняется по адресу http://localhost:8000
// или http://127.0.0.1:8000

// Сначала необходимо загрузить используемые модули
var http = require('http');     // API HTTP-сервера
var fs = require('fs');         // Для работы с локальными файлами

var server = new http.Server(); // Создать новый HTTP-сервер
server.listen(8000);            // Прослушивать порт 8000.

// Для регистрации обработчиков событий в Node используется метод "on()".
// Когда сервер получает новый запрос, для его обработки вызывается функция.
server.on("request", function (request, response) {
    // Выполнить разбор адреса URL
    var url = require('url').parse(request.url);

    // Специальный адрес URL, который вынуждает сервер выполнить задержку перед ответом.
    // Это может быть полезно для имитации работы с медленным сетевым подключением.
    if (url.pathname === "/test/delay") {
        // Величина задержки определяется из строки запроса
        // или устанавливается равной 2000 миллисекунд
        var delay = parseInt(url.query) || 2000;
        // Установить код состояния и заголовки ответа
        response.writeHead(200,{"Content-Type": "text/plain; charset=UTF-8"});
        // Начать отправку ответа немедленно
    response.write("Задержка на " + delay + " миллисекунд...");
    // А затем завершить другой функцией, которая будет вызвана позже.
    setTimeout(function() {
        response.write("готово.");
        response.end();
    }, delay);
}
// Если запрошен адрес "/test/mirror", отправить запрос обратно целиком.
// Удобно, когда необходимо увидеть тело и заголовки запроса.
else if (url.pathname === "/test/mirror") {
    // Код состояния и заголовки ответа
    response.writeHead(200,{"Content-Type": "text/plain; charset=UTF-8"});
    // Вставить в ответ тело запроса
    response.write(request.method + " " + request.url +
                   " HTTP/" + request.httpVersion + "\r\n");
    // И заголовки запроса
    for(var h in request.headers) {
        response.write(h + ": " + request.headers[h] + "\r\n");
    }
    response.write("\r\n"); // За заголовками следует дополнительная пустая строка
    // Завершение отправки ответа выполняется следующими функциями-обработчиками:
    // Если в chunk передается тело запроса, вставить его в ответ.
    request.on("data", function(chunk) { response.write(chunk); });
    // Когда достигнут конец запроса, ответ также завершается.
    request.on("end", function(chunk) { response.end(); });
}
// Иначе обслужить файл из локального каталога.
else {
    // Получить имя локального файла и определить тип его содержимого по расширению.
    var filename = url.pathname.substring(1); // удалить начальный /
    var type;
    switch(filename.substring(filename.lastIndexOf(".")+1)) { // расшир.
    case "html":
    case "htm":      type = "text/html; charset=UTF-8"; break;
    case "js":       type = "application/JavaScript;charset=UTF-8"; break;
    case "css":      type = "text/css; charset=UTF-8"; break;
    case "txt" :     type = "text/plain; charset=UTF-8"; break;
    case "manifest": type = "text/cache-manifest; charset=UTF-8"; break;
    default:         type = "application/octet-stream"; break;
    }

    // Прочитать файл в асинхронном режиме и передать его содержимое единым блоком
    // в функцию обратного вызова. Для очень больших файлов лучше было бы
    // использовать API потоков ввода/вывода с функцией fs.createReadStream().
    fs.readFile(filename, function(err, content) {
        if (err) { // Если по каким-то причинам невозможно прочитать файл
            response.writeHead(404, {    // Отправить 404 Not Found
                "Content-Type": "text/plain; charset=UTF-8"});
            response.write(err.message); // Тело сообщения об ошибке
            response.end();              // Завершить отправку
        }
        else { // Иначе, если файл успешно прочитан.
            response.writeHead(200, // Установить код состояния и тип MIME
                               {"Content-Type": type});
                response.write(content); // Отправить содержимое файла
                response.end();          // И завершить отправку
            }
        });
    }
});
```

12.2.2. Пример использования Node: модуль утилит клиента HTTP

В примере 12.3 определяется несколько вспомогательных клиентских функций, использующих модуль «http», позволяющих выполнять GET- и POST-запросы протокола HTTP. Пример оформлен как модуль «httputils», который можно использовать в собственных программах, например:

```
var httputils = require("./httputils"); // Отметьте отсутствие расш. ".js"
httputils.get(url, function(status, headers, body) { console.log(body); });
```

При выполнении программного кода модуля функция `require()` **не использует обычную функцию** `eval()`**. Модули выполняются в специальном окружении, чтобы они не могли определять глобальные переменные или как-то иначе изменять глобальное пространство имен. Это специализированное окружение для выполнения модулей всегда включает глобальный объект с именем** `exports`**. Модули экспортируют свои API, определяя свойства этого объекта.**[1]

Пример 12.3. Модуль «httputils» для интерпретатора Node

```
//
// Модуль "httputils" для интерпретатора Node.
//

// Выполняет асинхронный HTTP GET-запрос для указанного URL и передает
// код состояния HTTP, заголовки и тело ответа указанной функции обратного
// вызова. Обратите внимание, как этот метод экспортируется через обьект exports.
exports.get = function(url, callback) {
    // Разобрать URL и получить необходимые фрагменты
    url = require('url').parse(url);
    var hostname = url.hostname, port = url.port || 80;
    var path = url.pathname, query = url.query;
    if (query) path += "?" + query;

    // Выполняем простой GET-запрос
    var client = require("http").createClient(port, hostname);
    var request = client.request("GET", path, {
        "Host": hostname            // Заголовки запроса
    });
    request.end();

    // Функция обработки ответа после начала его получения
    request.on("response", function(response) {
        // Указать кодировку, чтобы тело возвращалось как строка, а не байты
```

---

[1] **Интерпретатор Node реализует протокол CommonJS работы с модулями, описание которого можно найти по адресу *http://www.commonjs.org/specs/modules/1.0/*.**

```
        response.setEncoding("utf8");
        // Для сохранения тела ответа по мере получения
        var body = ""
        response.on("data", function(chunk) { body += chunk; });
        // По окончании тела ответа вызвать функцию обратного вызова
        response.on("end", function() {
            if (callback)
                callback(response.statusCode, response.headers, body);
        });
    });
};

// Простой HTTP POST-запрос с данными в теле запроса
exports.post = function(url, data, callback) {
    // Разобрать URL и получить необходимые фрагменты
    url = require('url').parse(url);
    var hostname = url.hostname, port = url.port || 80;
    var path = url.pathname, query = url.query;
    if (query) path += "?" + query;

    // Определить тип данных, отправляемых в теле запроса
    var type;
    if (data == null) data = "";
    if (data instanceof Buffer)                     // Двоичные данные
        type = "application/octet-stream";
    else if (typeof data === "string")              // Строковые данные
        type = "text/plain; charset=UTF-8";
    else if (typeof data === "object") {            // Пары имя/значение
        data = require("querystring").stringify(data);
        type = "application/x-www-form-urlencoded";
    }

    // Выполнить POST-запрос и отправить тело запроса
    var client = require("http").createClient(port, hostname);
    var request = client.request("POST", path, {
        "Host": hostname,
        "Content-Type": type
    });
    request.write(data);                            // Отправить тело запроса
    request.end();
    request.on("response", function(response) {     // Обработать ответ
        response.setEncoding("utf8");               // Предполагается текст
        var body = ""                               // Для хранения тела ответа
        response.on("data", function(chunk) { body += chunk; });
        response.on("end", function() {             // По завершении вызвать обработчик
            if (callback)
                callback(response.statusCode, response.headers, body);
        });
    });
};
```

II

Клиентский JavaScript

В данной части книги в главах с 13 по 22 язык JavaScript описан в том виде, в котором он реализован в веб-броузерах. В этих главах вводится много новых JavaScript-объектов, представляющих окна, документы и содержимое документов в веб-броузерах. В них также описываются важные прикладные интерфейсы для организации сетевых взаимодействий, сохранения и извлечения данных и рисования графических изображений в веб-приложениях.

- Глава 13 «JavaScript в веб-броузерах»
- Глава 14 «Объект Window»
- Глава 15 «Работа с документами»
- Глава 16 «Каскадные таблицы стилей»
- Глава 17 «Обработка событий»
- Глава 18 «Работа с протоколом HTTP»
- Глава 19 «Библиотека jQuery»
- Глава 20 «Сохранение данных на стороне клиента»
- Глава 21 «Работа с графикой и медиафайлами на стороне клиента»
- Глава 22 «Прикладные интерфейсы HTML5»

13

JavaScript в веб-броузерах

Первая часть этой книги была посвящена базовому языку JavaScript. Теперь мы перейдем к тому языку JavaScript, который используется в веб-броузерах и обычно называется клиентским JavaScript (client-side JavaScript). Большинство примеров, которые мы видели до сих пор, будучи корректным JavaScript-кодом, не зависели от какого-либо контекста; это были JavaScript-фрагменты, не ориентированные на запуск в какой-либо определенной среде. Эта глава описывает такой контекст.

До обсуждения JavaScript давайте поближе познакомимся с веб-страницами, отображаемыми в веб-броузерах. Некоторые страницы представляют статическую информацию; их можно называть документами. (Представление такой статической информации может быть дополнено динамическим поведением с помощью JavaScript, но сама по себе информация остается статической.) Другие страницы больше похожи на приложения, чем на документы. Эти страницы способны динамически загружать новую информацию по мере необходимости, они могут состоять преимущественно из графических изображений, могут действовать без подключения к серверу, сохранять данные локально и, как следствие, способны восстанавливать свое состояние при повторном посещении. Имеются также веб-страницы, которые занимают промежуточное положение и объединяют в себе особенности документов и приложений.

Эта глава начинается с краткого обзора клиентского JavaScript. Она представляет простой пример и описание роли JavaScript в веб-документах и в веб-приложениях. Кроме того, здесь приводится краткое содержание глав второй части книги. Последующие разделы описывают, как программный код на языке JavaScript встраивается в HTML-документы и выполняется в них. Далее следуют разделы с обсуждением трех важных тем программирования на JavaScript: совместимости, удобства и безопасности.

13.1. Клиентский JavaScript

Объект `Window` является средоточием всех особенностей и прикладных интерфейсов клиентского JavaScript. Он представляет окно веб-броузера или фрейм, а сослаться на него можно с помощью идентификатора `window`. Объект `Window` определяет свойства, такие как `location`, которое ссылается на объект `Location`, определяющий URL текущего окна, и позволяет сценарию загружать в окно содержимое других адресов URL:

```
// Установить значение свойства для переход на новую веб-страницу
window.location = "http://www.oreilly.com/";
```

Кроме того, объект `Window` определяет методы, такие как `alert()`, который отображает диалог с сообщением, и `setTimeout()`, который регистрирует функцию для вызова через указанный промежуток времени:

```
// Ждать 2 секунды и вывести диалог с приветствием
setTimeout(function() { alert("Привет, Мир!"); }, 2000);
```

Обратите внимание, что программный код выше неявно использует свойство объекта `window`. Объект `Window` в клиентском JavaScript является глобальным объектом. Это означает, что объект `Window` находится на вершине цепочки областей видимости и что его свойства и методы фактически являются глобальными переменными и функциями. Объект `Window` имеет свойство `window`, которое всегда ссылается на сам объект. Это свойство можно использовать для ссылки на сам объект, но обычно в этом нет необходимости, если требуется просто сослаться на свойство глобального объекта окна.

Объект `Window` определяет также множество других важных свойств, методов и конструкторов. Полный их перечень можно найти в главе 14.

Одним из наиболее важных свойств объекта `Window` является свойство `document`: оно ссылается на объект `Document`, который представляет содержимое документа, отображаемого в окне. Объект `Document` имеет важные методы, такие как `getElementById()`, который возвращает единственный элемент документа (представляющий пару из открывающего и закрывающего тегов HTML и все, что содержится между ними), опираясь на значение атрибута `id` элемента:

```
// Отыскать элемент с атрибутом id="timestamp"
var timestamp = document.getElementById("timestamp");
```

Объект `Element`, возвращаемый методом `getElementById()`, также имеет ряд важных свойств и методов, позволяющих сценарию извлекать содержимое элемента, устанавливать значения его атрибутов и т. д.:

```
// Если элемент пуст, вставить в него текущую дату и время
if (timestamp.firstChild == null)
    timestamp.appendChild(document.createTextNode(new Date().toString()));
```

Приемы получения ссылок на элементы, обхода элементов и изменения содержимого документа описываются в главе 15.

Каждый объект `Element` имеет свойства `style` и `className`, позволяющие определять стили CSS элемента документа или изменять имена классов CSS, применяемых к элементу. Установка этих свойств, имеющих отношение к CSS, изменяют визуальное представление элемента документа:

```
// Явно изменить представление элемента заголовка
timestamp.style.backgroundColor = "yellow";

// Или просто изменить класс и позволить определять особенности
// визуального представления с помощью каскадных таблиц стилей:
timestamp.className = "highlight";
```

Свойства `style` и `className`, а также другие приемы управления стилями CSS описываются в главе 16.

Другим важным множеством свойств объектов `Window`, `Document` и `Element` являются свойства, ссылающиеся на обработчики событий. Они позволяют сценариям определять функции, которые должны вызываться асинхронно при возникновении определенных событий. Обработчики событий позволяют программам на языке JavaScript изменять поведение окон, документов и элементов, составляющих документы. Свойства, ссылающиеся на обработчики событий, имеют имена, начинающиеся с «on», и могут использоваться, как показано ниже:

```
// Обновить содержимое элемента timestamp, когда пользователь щелкнет на нем
timestamp.onclick = function() { this.innerHTML = new Date().toString(); }
```

Одним из наиболее важных обработчиков событий является обработчик `onload` объекта `Window`. Это событие возникает, когда содержимое документа, отображаемое в окне, будет загружено полностью и станет доступно для выполнения манипуляций. Программный код на языке JavaScript обычно заключается в обработчик события `onload`. События являются темой главы 17. Пример 13.1 использует обработчик `onload` и демонстрирует дополнительные приемы получения ссылок на элементы документа, изменения классов CSS и определения обработчиков других событий в клиентском JavaScript. В этом примере программный код на языке JavaScript заключен в HTML-тег `<script>`. Подробнее этот тег описывается в разделе 13.2. Обратите внимание, что в этом примере имеется определение функции, заключенное в определение другой функции. Вложенные функции часто используются в клиентских сценариях на языке JavaScript, особенно в виде обработчиков событий.

Пример 13.1. Простой клиентский сценарий на языке JavaScript, исследующий содержимое документа

```
<!DOCTYPE html>
<html>
<head>
<style>
/* Стили CSS для этой страницы */
.reveal * { display: none; }       /* Элементы с атрибутом class="reveal" невидимы */
.reveal *.handle { display: block;} /* Кроме элементов с class="handle" */
</style>
<script>
// Ничего не делать, пока документ не будет загружен полностью
window.onload = function() {
    // Отыскать все контейнерные элементы с классом "reveal"
    var elements = document.getElementsByClassName("reveal");
    for(var i = 0; i < elements.length; i++) { // Для каждого такого элемента...
        var elt = elements[i];
        // Отыскать элементы с классом "handle" в контейнере
        var title = elt.getElementsByClassName("handle")[0];
        // После щелчка на этом элементе сделать видимым остальное содержимое
        addRevealHandler(title, elt);
    }

    function  addRevealHandler(title, elt) {
        title.onclick = function() {
            if (elt.className == "reveal") elt.className = "revealed";
            else if (elt.className == "revealed") elt.className = "reveal";
            }
        }
    };
    </script>
    </head>
    <body>
    <div class="reveal">
    <h1 class="handle">Щелкните здесь, чтобы увидеть скрытый текст</h1>
    <p>Этот абзац невидим. Он появляется после щелчка на заголовке.</p>
    </div>
    </body>
    </html>
```

Во введении к этой главе отмечалось, что некоторые веб-страницы ведут себя как документы, а некоторые – как приложения. Следующие два подраздела исследуют применение JavaScript в обоих типах веб-страниц.

13.1.1. Сценарии JavaScript в веб-документах

Программы на языке JavaScript могут манипулировать *содержимым* документа через объект `Document` и содержащиеся в нем объекты `Element`. Они могут изменять *визуальное представление* содержимого, управляя стилями и классами CSS, и определять *поведение* элементов документа, регистрируя соответствующие обработчики событий. Комбинация управляемого содержимого, представления и поведения называется динамическим HTML (Dynamic HTML, или DHTML), а приемы создания документов DHTML описываются в главах 15, 16 и 17.

Программный код на языке JavaScript в веб-документах обычно должен использоваться ограниченно и выполнять определенную роль. Основная цель использования JavaScript – облегчить пользователю получение или отправку информации. Работа пользователя не должна зависеть от наличия поддержки JavaScript в броузере; сценарии на JavaScript можно отнести к подручным средствам, которые:

- Создают анимационные и другие визуальные эффекты, ненавязчиво руководя действиями пользователя и помогая ему передвигаться по странице.
- Сортируют столбцы таблиц, упрощая пользователю поиск требуемой информации.
- Скрывают определенное содержимое и отображают детали по мере «погружения» пользователя в содержимое.

13.1.2. Сценарии JavaScript в веб-приложениях

Веб-приложения применяют все возможности JavaScript и DHTML, которые используются в веб-документах, но помимо управления содержимым, его представлением и поведением они также используют преимущества других фундаментальных механизмов, предоставляемых веб-броузерами.

Чтобы понять суть веб-приложений, важно осознать, что веб-броузеры развивались не только как инструменты для отображения документов и давно уже трансформировались в некоторое подобие простых операционных систем. Сравните: традиционные операционные системы позволяют создавать ярлыки (представляющие файлы и приложения) на рабочем столе и в папках. Веб-броузеры позволяют создавать закладки (представляющие документы и веб-приложения) на панели инструментов и в папках. Операционные системы выполняют множество приложений в отдельных окнах; веб-броузеры отображают множество документов (или приложений) в отдельных вкладках. Операционные системы определяют низкоуровневые API для организации сетевых взаимодействий, рисования графики и сохранения файлов. Веб-броузеры определяют низкоуровневые API для организации сетевых взаимодействий (глава 18), сохранения данных (глава 20) и рисования графики (глава 21).

Представляя веб-броузеры как упрощенные операционные системы, веб-приложения можно определить как веб-страницы, в которых используется программный код на языке JavaScript для доступа к расширенным механизмам (таким как сетевые взаимодействия, рисование графики и сохранение данных) броузеров. Самым известным из этих механизмов является объект `XMLHttpRequest`, который обеспечивает сетевые взаимодействия посредством управляемых HTTP-запросов. Веб-приложения используют этот механизм для получения новой информации с сервера без полной перезагрузки страницы. Веб-приложения, использующие этот прием, часто называют Ajax-приложениями, и они образуют фундамент того, что известно под названием «Web 2.0». Объект `XMLHttpRequest` во всех подробностях рассматривается в главе 18.

Спецификация HTML5 (которая на момент написания этих строк еще находилась в состоянии проекта) и ряд связанных с ней спецификаций определяют ряд других важных прикладных интерфейсов для веб-приложений. В их число входят прикладные интерфейсы для сохранения данных и рисования графики, которые описываются в главах 21 и 20, а также множество других возможностей, таких как геопозиционирование (geolocation), управление журналами посещений и фоновые потоки выполнения. Когда все эти прикладные интерфейсы будут реализованы, они обеспечат дальнейшее расширение возможностей веб-приложений. Подробнее о них рассказывается в главе 22.

Безусловно, сценарии на языке JavaScript занимают в веб-приложениях более важное положение, чем в веб-документах. Сценарии на JavaScript расширяют возможности веб-документов, но при правильном оформлении документы остаются полностью доступными даже при отключенной поддержке JavaScript. Веб-приложения по определению являются программами на языке JavaScript, использующими механизмы, предоставляемые веб-броузерами, и не будут работать при отключенной поддержке JavaScript.[1]

13.2. Встраивание JavaScript-кода в разметку HTML

Клиентский JavaScript-код может встраиваться в HTML-документы четырьмя способами:

---

[1] Интерактивные веб-страницы, взаимодействующие с серверными CGI-сценариями посредством форм HTML также можно считать «веб-приложениями», и они могут быть написаны без применения JavaScript. Однако это не тот тип веб-приложений, которые мы будем обсуждать в этой книге.

- встроенные сценарии между парой тегов `<script>` и `</script>`;
- из внешнего файла, заданного атрибутом `src` тега `<script>`;
- в обработчик события, заданный в качестве значения HTML-атрибута, такого как `onclick` или `onmouseover`;
- как тело URL-адреса, использующего специальный спецификатор псевдопротокола `JavaScript:`.

В следующих далее подразделах описываются все четыре способа встраивания программного кода на языке JavaScript. Следует отметить, что HTML-атрибуты обработчиков событий и адреса URL с псевдопротоколом `javascript:` редко используются в современной практике программирования на языке JavaScript (они были более распространены на раннем этапе развития Всемирной паутины). Встроенные сценарии (в тегах `<script>` без атрибута `src`) также стали реже использоваться по сравнению с прошлым. Согласно философии программирования, известной как *ненавязчивый JavaScript* (unobtrusive JavaScript), содержимое (разметка HTML) и поведение (программный код на языке JavaScript) должны быть максимально отделены друг от друга. Следуя этой философии программирования, сценарии на языке JavaScript лучше встраивать в HTML-документы с помощью элементов `<script>`, имеющих атрибут `src`.

13.2.1. Элемент <script>

Клиентские JavaScript-сценарии могут встраиваться в HTML-файлы между тегами `<script>` и `</script>`:

```
<script>
// Здесь располагается JavaScript-код
</script>
```

В языке разметки XHTML содержимое тега `<script>` обрабатывается наравне с содержимым любого другого тега. Если JavaScript-код содержит символы `<` или `&`, они интерпретируются как элементы XML-разметки. Поэтому в случае применения языка XHTML лучше помещать весь JavaScript-код внутрь секции CDATA:

```
<script><![CDATA[
// Здесь располагается JavaScript-код
]]></script>
```

В примере 13.2 демонстрируется содержимое HTML-файла, включающего простую программу на языке JavaScript. Действия программы описываются в комментариях, тем не менее замечу, что главная цель этого примера в том, чтобы продемонстрировать, как JavaScript-код встраивается в файлы HTML наряду со всем остальным, в данном случае – со стилями CSS. Обратите внимание, что этот пример по своей структуре напоминает пример 13.1 и точно так же использует обработчик события `onload`.

Пример 13.2. Простые часы с цифровым табло на JavaScript

```
<!DOCTYPE html>    <!-- Это файл HTML5 -->
<html>             <!-- Корневой элемент -->
<head>             <!-- Заголовок, здесь располагаются сценарии и стили -->
<title>Digital Clock</title>
<script>                     // Сценарий на JavaScript
```

```
// Определение функции для отображения текущего времени
function displayTime() {
    var elt = document.getElementById("clock"); // Найти элемент с id="clock"
    var now = new Date();                       // Получить текущее время
    elt.innerHTML = now.toLocaleTimeString();   // Отобразить его
    setTimeout(displayTime, 1000);              // Вызвать снова через 1 сек.
}
window.onload = displayTime; // Начать отображение времени после загрузки документа.
</script>
<style>                     /* Таблица стилей CSS для часов */
#clock {                    /* Стили применяются к элементу с id="clock" */
  font: bold 24pt sans;     /* Использовать большой полужирный шрифт */
  background: #ddf;         /* Светлый, голубовато-серый фон */
  padding: 10px;            /* Отступы вокруг */
  border: solid black 2px;  /* И сплошная черная рамка */
  border-radius: 10px;      /* Закругленные углы (если поддерживаются) */
}
</style>
</head>
<body>                     <!-- Тело - отображаемая часть документа -->
<h1>Цифровые часы</h1>     <!-- Вывести заголовок -->
<span id="clock"></span>   <!-- Время выводится здесь -->
</body>
</html>
```

13.2.2. Сценарии во внешних файлах

Тег `<script>` поддерживает атрибут `src`, который определяет URL-адрес файла, содержащего JavaScript-код. Используется он следующим образом:

```
<script src="../../scripts/util.js"></script>
```

Файл JavaScript-кода обычно имеет расширение *.js* и содержит JavaScript-код в «чистом виде» без тегов `<script>` или любого другого HTML-кода.

Тег `<script>` с атрибутом `src` ведет себя точно так, как если бы содержимое указанного файла JavaScript-кода находилось непосредственно между тегами `<script>` и `</script>`. Обратите внимание, что закрывающий тег `</script>` обязателен, даже когда указан атрибут `src` и между тегами отсутствует JavaScript-код. В разметке XHTML в подобных случаях можно использовать единственный тег `<script/>`.

При использовании атрибута `src` любое содержимое между открывающим и закрывающим тегами `<script>` игнорируется. При желании в качестве содержимого в тег `<script>` можно вставлять описание включаемого программного кода или информацию об авторском праве. Однако следует заметить, что инструменты проверки соответствия разметки требованиям стандарта HTML5 будут выдавать предупреждения, если между тегами `<script src="">` и `</script>` будет находиться какой-либо текст, не являющийся пробельными символами или комментариями на языке JavaScript.

Использование тега с атрибутом `src` дает ряд преимуществ:

- HTML-файлы становятся проще, т. к. из них можно убрать большие блоки JavaScript-кода, что помогает отделить содержимое от поведения.

- JavaScript-функцию или другой JavaScript-код, используемый несколькими HTML-файлами, можно держать в одном файле и считывать при необходимости. Это уменьшает объем занимаемой дисковой памяти и намного облегчает поддержку программного кода, т. к. отпадает необходимость править каждый HTML-файл при изменении кода.
- Если сценарий на языке JavaScript используется сразу несколькими страницами, он будет загружаться броузером только один раз, при первом его использовании – последующие страницы будут извлекать его из кэша броузера.
- Атрибут `src` принимает в качестве значения произвольный URL-адрес, поэтому JavaScript-программа или веб-страница с одного веб-сервера может воспользоваться кодом (например, из библиотеки подпрограмм), предоставляемым другими веб-серверами. Многие рекламодатели в Интернете используют этот факт.
- Возможность загружать сценарии с других сайтов еще больше увеличивает выгоды, получаемые от кэширования: компания Google продвигает использование стандартных, хорошо известных URL-адресов для часто используемых клиентских библиотек, что позволяет броузерам хранить в кэше единственную копию, совместно используемую многими сайтами в Веб. Привязка сценариев JavaScript к серверам компании Google может существенно уменьшить время запуска веб-страниц, поскольку библиотека наверняка уже будет храниться в кэше броузера пользователя, но при этом вы должны доверять стороннему программному коду, который может оказаться критически важным для вашего сайта. За дополнительной информацией обращайтесь по адресу: *http://code.google.com/apis/ajaxlibs/*.

Возможность загрузки сценариев со сторонних серверов, отличных от тех, где находятся документы, использующие эти сценарии, влечет за собой важное следствие, имеющее отношение к обеспечению безопасности. Политика общего происхождения, описываемая в разделе 13.6.2, предотвращает возможность взаимодействия сценария на JavaScript в документе из одного домена с содержимым из другого домена. Однако следует отметить, что источник получения самого сценария не имеет значения, значение имеет источник получения документа, в который встраивается сценарий. Таким образом, политика общего происхождения в данном случае неприменима: JavaScript-код может взаимодействовать с документами, в которые он встраивается, даже если этот код получен из другого источника, нежели сам документ. Включая сценарий в свою веб-страницу с помощью атрибута `src`, вы предоставляете автору сценария (или веб-мастеру домена, откуда загружается сценарий) полный контроль над своей веб-страницей.

13.2.3. Тип сценария

JavaScript изначально был языком сценариев для Всемирной паутины, и по умолчанию предполагалось, что элементы `<script>` содержат или ссылаются на программный код на языке JavaScript. Если у вас появится необходимость использовать нестандартный язык сценариев, такой как VBScript корпорации Microsoft (который поддерживается только в Internet Explorer), необходимо в атрибуте `type` указать MIME-тип сценария:

```
<script type="text/vbscript">
  ' // Здесь располагается VBScript-код
</script>
```

По умолчанию атрибут `type` получает значение «text/JavaScript». При желании можно явно указать это значение, однако в этом нет необходимости.

В старых броузерах вместо атрибута `type` использовался атрибут `language` тега `<script>`, и вы по-прежнему можете встретить веб-страницы, включающие такие теги:

```
<script language="javascript">
// Здесь располагается JavaScript-код...
</script>
```

Атрибут `language` считается устаревшим и не должен более использоваться.

Когда веб-броузер встречает элемент `<script>` с атрибутом `type`, значение которого он не может распознать, он пытается проанализировать элемент, но не отображает и не выполняет его содержимое. Это означает, что элемент `<script>` можно использовать для встраивания в документ произвольных текстовых данных: достаточно просто указать значение атрибута `type`, указывающее, что данные не являются выполняемым программным кодом. Чтобы извлечь эти данные, можно воспользоваться свойством `text` объекта `HTMLElement`, представляющего элемент `script` (как получить эти элементы, описывается в главе 15). Однако важно отметить, что такой прием встраивания данных работает только при непосредственном встраивании их в разметку. Если указать атрибут `src` и неизвестное значение в атрибуте `type`, броузер проигнорирует этот тег и ничего не будет загружать с указанного адреса URL.

13.2.4. Обработчики событий в HTML

JavaScript-код, расположенный в теге `<script>`, исполняется один раз, когда содержащий его HTML-файл считывается в веб-броузер. Для обеспечения интерактивности программы на языке JavaScript должны определять обработчики событий – JavaScript-функции, которые регистрируются в веб-броузере и автоматически вызываются веб-броузером в ответ на определенные события (такие как ввод данных пользователем). Как было показано в начале этой главы, JavaScript-код может регистрировать обработчики событий, присваивая функции свойствам объектов `Element` (таким как `onclick` или `onmouseover`), представляющих HTML-элементы в документе. (Существует и другой способ регистрации обработчиков событий – подробности приводятся в главе 17.)

Свойства обработчиков событий, такие как `onclick`, отражают HTML-атрибуты с теми же именами, что позволяет определять обработчики событий, помещая JavaScript-код в HTML-атрибуты. Например, чтобы определить обработчик события, который вызывается, когда пользователь щелкает на флажке в форме, код обработчика указывается в качестве значения атрибута `onchange` HTML-элемента, определяющего флажок:

```
<input type="checkbox" name="options" value="giftwrap"
       onchange="order.options.giftwrap = this.checked;">
```

Обратите внимание на атрибут `onchange`. JavaScript-код, являющийся значением этого атрибута, будет выполняться всякий раз, когда пользователь будет щелкать на флажке.

Атрибуты обработчиков событий, включенных в разметку HTML, могут содержать одну или несколько JavaScript-инструкций, отделяемых друг от друга точками с запятой. Эти инструкции будут преобразованы интерпретатором в тело функции, которая в свою очередь станет значением соответствующего свойства обработчика события. (Подробное описание, как выполняется преобразование текстового содержимого HTML-атрибутов в функции на языке JavaScript, приводится в разделе 17.2.2.) Однако обычно в HTML-атрибуты обработчиков событий включаются простые инструкции присваивания, как в примере выше, или простые вызовы функций, объявленных где-то в другом месте. Это позволяет держать большую часть JavaScript-кода внутри сценариев и ограничивает степень взаимопроникновения JavaScript- и HTML-кода. На практике многие веб-разработчики считают плохим стилем использование HTML-атрибутов обработчиков событий и предпочитают отделять содержимое от поведения.

13.2.5. JavaScript в URL

Еще один способ выполнения JavaScript-кода на стороне клиента – включение этого кода в URL-адресе вслед за спецификатором псевдопротокола `javascript:`. Этот специальный тип протокола обозначает, что тело URL-адреса представляет собою произвольный JavaScript-код, который должен быть выполнен интерпретатором JavaScript. Он интерпретируется как единственная строка, и потому инструкции в ней должны быть отделены друг от друга точками с запятой, а для комментариев следует использовать комбинации символов /* */, а не //. «Ресурсом», который определяется URL-адресом `javascript:`, является значение, возвращаемое этим программным кодом, преобразованное в строку. Если программный код возвращает значение `undefined`, считается, что ресурс не имеет содержимого.

URL вида `javascript:` можно использовать везде, где допускается указывать обычные URL: в атрибуте `href` тега `<a>`, в атрибуте `action` тега `<form>` и даже как аргумент метода, такого как `window.open()`. Например, адрес URL с программным кодом на языке JavaScript в гиперссылке может иметь такой вид:

```
<a href="JavaScript:new Date().toLocaleTimeString();">
Который сейчас час?
</a>
```

Некоторые броузеры (такие как Firefox) выполняют программный код в URL и используют возвращаемое значение в качестве содержимого нового отображаемого документа. Точно так же, как при переходе по ссылке `http:`, броузер стирает текущий документ и отображает новое содержимое. Значение, возвращаемое примером выше, не содержит HTML-теги, но если бы они имелись, броузер мог бы отобразить их точно так же, как любой другой HTML-документ, загруженный в броузер. Другие броузеры (такие как Chrome и Safari) не позволяют URL-адресам, как в примере выше, затирать содержимое документа – они просто игнорируют возвращаемое значение. Однако они поддерживают URL-адреса вида:

```
<a href="JavaScript:alert(new Date().toLocaleTimeString());">
Узнать время, не затирая документ
</a>
```

Когда загружается такой URL-адрес, броузер выполняет JavaScript-код, но, т.к. он не имеет возвращаемого значения (метод `alert()` возвращает значение `undefined`), такие броузеры, как Firefox, не затирают текущий отображаемый документ. (В данном случае URL-адрес `javascript:` служит той же цели, что и обработчик события `onclick`. Ссылку выше лучше было бы выразить как обработчик события `onclick` элемента `<button>` – элемент `<a>` в целом должен использоваться только для гиперссылок, которые загружают новые документы.) Если необходимо гарантировать, что URL-адрес `javascript:` не затрет документ, можно с помощью оператора `void` обеспечить принудительный возврат значения `undefined`:

```
<a href="javascript:void window.open('about:blank');">Open Window</a>
```

Без оператора `void` в этом URL-адресе значение, возвращаемое методом `Window.open()`, было бы преобразовано в строку и (в некоторых броузерах) текущий документ был бы затерт новым документом с текстом:

```
[object Window]
```

Подобно HTML-атрибутам обработчиков событий, URL-адреса `javascript:` являются пережитком раннего периода развития Веб и не должны использоваться в современных HTML-страницах. URL-адреса `javascript:` могут сослужить полезную службу, если использовать их вне контекста HTML-документов. Если потребуется проверить работу небольшого фрагмента JavaScript-кода, можно ввести URL-адрес `javascript:` непосредственно в адресную строку броузера. Другое узаконенное применение URL-адресов `javascript:` – создание закладок в броузерах, как описывается ниже.

13.2.5.1. Букмарклеты

«Закладкой» в веб-броузере называется сохраненный URL-адрес. Если закладка содержит URL-адрес `javascript:`, такая закладка играет роль мини-программы на языке JavaScript, которая называется *букмарклетом* (*bookmarklet*). Букмарклеты легко можно запустить из меню или панели инструментов. Программный код в букмарклете выполняется, как если бы он являлся сценарием в странице; он может читать и изменять содержимое документа, его представление и поведение. Если букмарклет не возвращает какое-либо значение, он может манипулировать содержимым любого отображаемого документа, не замещая его новым содержимым.

Взгляните на следующий фрагмент URL `javascript:` в теге `<a>`. Щелчок на ссылке открывает простейший обработчик JavaScript-выражений, который позволяет вычислять выражения и выполнять инструкции в контексте страницы:

```
<a href='javascript:
var e = "", r = ""; /* Вычисляемое выражение и результат */
do {
    /* Отобразить выражение и результат, а затем запросить новое выражение */
    e = prompt("Выражение: " + e + "\n" + r + "\n", e);
    try { r = "Результат: " + eval(e); } /* Попробовать вычислить выражение */
    catch(ex) { r = ex; }                /* Или запомнить ошибку */
} while(e); /* продолжать, пока не будет введено пустое выражение */
            /* или не будет выполнен щелчок на кнопке Отмена*/
void 0;     /* Это предотвращает затирание текущего документа */
'>
Обработчик JavaScript-выражений
</a>
```

Обратите внимание: несмотря на то что этот программный код записан в нескольких строках, синтаксический анализатор разметки HTML обработает его как одну строку, а потому однострочные комментарии (//) здесь работать не будут. Кроме того, не забывайте, что весь этот программный код является частью значения HTML-атрибута, заключенного в одиночные кавычки, поэтому этот программный код не может содержать одиночные кавычки.

Ссылки, подобные этой, удобны, когда они «зашиты» в тело разрабатываемой страницы, но еще более удобны, когда они хранятся как закладки, которые можно запустить из любой страницы. Обычно закладки создаются щелчком правой кнопкой мыши на странице и выбором в контекстном меню пункта Добавить страницу в закладки или подобного ему. В броузере Firefox для этого достаточно просто перетащить ссылку на панель закладок.

13.3. Выполнение JavaScript-программ

Вообще говоря, не существует формального определения *программы* на клиентском языке JavaScript. Можно лишь сказать, что программой является весь программный код на языке JavaScript, присутствующий в веб-странице (встроенные сценарии, обработчики событий в разметке HTML и URL-адреса `javascript:`), а также внешние сценарии JavaScript, на которые ссылаются атрибуты `src` тегов `<script>`. Все эти отдельные фрагменты программного кода совместно используют один и тот же глобальный объект `Window`. Это означает, что все они видят один и тот же объект `Document` и совместно используют один и тот же набор глобальных функций и переменных: если сценарий определяет новую глобальную переменную или функцию, эта переменная или функция будет доступна любому программному коду на языке JavaScript, который будет выполняться после этого сценария.

Если веб-страница содержит встроенный фрейм (элемент `<iframe>`), JavaScript-код во встроенном документе будет работать с другим глобальным объектом, отличным от глобального объекта в объемлющем документе, и его можно рассматривать как отдельную JavaScript-программу. Однако напомню, что не существует формального определения, устанавливающего границы JavaScript-программы. Если оба документа, вмещающий и вложенный, получены с одного сервера, то программный код в одном документе сможет взаимодействовать с программным кодом в другом документе и их можно считать взаимодействующими частями одной программы. Подробнее о глобальном объекте `Window` и о взаимодействии программ, выполняющихся в разных окнах и фреймах, рассказывается в разделе 14.8.3.

URL-адреса `javascript:` в букмарклетах существуют за пределами какого-либо документа, и их можно рассматривать, как своего рода пользовательские расширения или дополнения к другим программам. Когда пользователь запускает букмарклет, программный код в букмарклете получает доступ к глобальному объекту и содержимому текущего документа и может манипулировать им как угодно.

Программы на языке JavaScript выполняются в два этапа. На первом этапе производится загрузка содержимого документа и запускается программный код в элементах `<script>` (и встроенные сценарии, и внешние). Обычно (но не всегда – подробнее об этом рассказывается в разделе 13.3.1) сценарии выполняются в порядке их следования в документе. Внутри каждого сценария программный код выполняется последовательно, от начала до конца, с учетом условных инструкций, циклов и других инструкций управления потоком выполнения.

После загрузки документа и выполнения всех сценариев начинается второй этап выполнения JavaScript-программы, асинхронный и управляемый событиями. На протяжении этого этапа, управляемого событиями, веб-броузер вызывает функции обработчиков (которые определены в HTML-атрибутах обработчиков событий, установлены сценариями, выполненными на первом этапе, или обработчиками событий, вызывавшимися ранее) в ответ на события, возникающие асинхронно. Обычно обработчики событий вызываются в ответ на действия пользователя (щелчок мышью, нажатие клавиши и т. д.), но могут также вызываться в ответ на сетевые взаимодействия, по истечении установленного промежутка времени или при возникновении ошибочных ситуаций в JavaScript-коде. Подробнее о событиях и обработчиках событий рассказывается в главе 17. Некоторая дополнительная информация о них будет также приводиться в разделе 13.3.2. Обратите внимание, что URL-адреса `javascript:`, встроенные в веб-страницу, также можно отнести к категории обработчиков событий, т. к. они не выполняются, пока не будут активированы действиями пользователя, такими как щелчок мышью на ссылке или отправка формы на сервер.

Одно из первых событий, возникающих на управляемом событиями этапе выполнения, является событие `load`, которое сообщает, что документ полностью загружен и готов к работе. JavaScript-программы нередко используют это событие как механизм запуска. На практике часто можно увидеть программы, сценарии которых определяют функции, но не выполняют никаких действий, кроме определения обработчика события `onload`, вызываемого по событию `load` и запускающего управляемый событиями этап выполнения. Именно обработчик события `onload` выполняет операции с документом и реализует все, что должна делать программа. Этап загрузки JavaScript-программы протекает относительно быстро, обычно он длится не более одной-двух секунд. Управляемый событиями этап выполнения, наступающий сразу после загрузки документа, длится на протяжении всего времени, пока документ отображается веб-броузером. Поскольку этот этап является асинхронным и управляемым событиями, он может состоять из длительных периодов отсутствия активности, когда не выполняется никакой программный код JavaScript, перемежающихся всплесками активности, вызванной действиями пользователя или событиями, связанными с сетевыми взаимодействиями. Подробнее оба этапа выполнения JavaScript-программ рассматриваются в разделе 13.3.4.

Обе разновидности языка, базовый JavaScript и клиентский JavaScript, поддерживают однопоточную модель выполнения. Сценарии и обработчики событий выполняются последовательно, не конкурируя друг с другом. Такая модель выполнения обеспечивает простоту программирования на языке JavaScript и обсуждается в разделе 13.3.3.

13.3.1. Синхронные, асинхронные и отложенные сценарии

Когда поддержка JavaScript впервые появилась в веб-броузерах, не существовало никаких инструментов обхода и управления структурой содержимого документа. Единственный способ, каким JavaScript-код мог влиять на содержимое документа, – это генерировать содержимое в процессе загрузки документа. Делалось это с помощью метода `document.write()`. В примере 13.3 показано, как выглядел ультрасовременный JavaScript-код в 1996 году.

Пример 13.3. Генерация содержимого документа во время загрузки

```
<h1>Таблица факториалов</h1>
<script>
function factorial(n) {    // Функция вычисления факториалов
    if (n <= 1) return n;
    else return n*factorial(n-1);
}

document.write("<table>");                          // Начало HTML-таблицы
document.write("<tr><th>n</th><th>n!</th></tr>"); // Вывести заголовок таблицы
for(var i = 1; i <= 10; i++) {                      // Вывести 10 строк
   document.write("<tr><td>" + i + "</td><td>" + factorial(i) + "</td></tr>");
}
document.write("</table>");                         // Конец таблицы
document.write("Generated at " + new Date());      // Вывести время
</script>
```

Когда сценарий передает текст методу `document.write()`, этот текст добавляется во входной поток документа, и механизм синтаксического анализа разметки HTML действует так, как если бы элемент `<script>` был замещен этим текстом. Использование метода `document.write()` более не считается хорошим стилем программирования, но его применение по-прежнему возможно (раздел 15.10.2), и этот факт имеет важное следствие. Когда механизм синтаксического анализа разметки HTML встречает элемент `<script>`, он должен, по умолчанию, выполнить сценарий, прежде чем продолжить разбор и отображение документа. Это не является проблемой для встроенных сценариев, но если сценарий находится во внешнем файле, на который ссылается атрибут `src`, это означает, что часть документа, следующая за сценарием, не появится в окне броузера, пока сценарий не будет загружен и выполнен.

Такой *синхронный*, или *блокирующий*, порядок выполнения действует только по умолчанию. Тег `<script>` может иметь атрибуты `defer` и `async`, которые (в броузерах, поддерживающих их) определяют иной порядок выполнения сценариев. Это логические атрибуты – они не имеют значения; они просто должны присутствовать в теге `<script>`. Согласно спецификации HTML5, эти атрибуты принимаются во внимание, только когда используются вместе с атрибутом `src`, однако некоторые броузеры могут поддерживать атрибут `defer` и для встроенных сценариев:

```
<script defer src="deferred.js"></script>
<script async src="async.js"></script>
```

Оба атрибута, `defer` и `async`, сообщают броузеру, что данный сценарий не использует метод `document.write()` и не генерирует содержимое документа, и что броузер может продолжать разбор и отображение документа, пока сценарий загружается. Атрибут `defer` заставляет броузер отложить выполнение сценария до момента, когда документ будет загружен, проанализирован и станет готов к выполнению операций. Атрибут `async` заставляет броузер выполнить сценарий, как только это станет возможно, но не блокирует разбор документа на время загрузки сценария. Если тег `<script>` имеет оба атрибута, броузер, поддерживающий оба этих атрибута, отдаст предпочтение атрибуту `async` и проигнорирует атрибут `defer`.

Обратите внимание, что отложенные сценарии выполняются в порядке их следования в документе. Асинхронные сценарии выполняются сразу же, как только будут загружены, т. е. они могут выполняться в произвольном порядке.

На момент написания этих строк атрибуты `async` и `defer` поддерживались не всеми броузерами, поэтому их следует рассматривать лишь как подсказки для оптимизации: веб-страницы должны проектироваться так, чтобы они продолжали корректно работать, даже если отложенные и асинхронные сценарии выполняются броузером синхронно.

Имеется возможность загружать и выполнять сценарии асинхронно, даже если броузер не поддерживает атрибут `async`, для чего достаточно динамически создать элемент `<script>` и вставить его в документ. Это действие реализует функция `loadasync()`, представленная в примере 13.4. Используемые ею приемы описываются в главе 15.

Пример 13.4. Асинхронная загрузка и выполнение сценария

```
// Асинхронная загрузка сценария из указанного URL-адреса и его выполнение
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; // Отыскать <head>
    var s = document.createElement("script");           // Создать элемент <script>
    s.src = url;                                        // Установить атрибут src
    head.appendChild(s);                                // Вставить <script> в <head>
}
```

Примечательно, что данная функция `loadasync()` загружает сценарии динамически – сценарии, не включенные в веб-страницу, и на которые отсутствуют статические ссылки из веб-страницы, загружаются в документ и становятся частью выполняемой JavaScript-программы.

13.3.2. Выполнение, управляемое событиями

Древняя JavaScript-программа, представленная в примере 13.3, является синхронной: она запускается на выполнение в процессе загрузки страницы, производит вывод и завершается. Такие программы редко используются в наши дни. Программы, которые пишутся в настоящее время, регистрируют функции обработчиков событий. Эти функции вызываются асинхронно, по событиям, для обработки которых они были зарегистрированы. Веб-приложения, в которых требуется реализовать поддержку горячих комбинаций клавиш для выполнения типичных операций, могут, например, регистрировать обработчики событий нажатия клавиш. Даже неинтерактивные программы используют события. Представьте, что требуется написать программу, которая должна проанализировать структуру документа и автоматически сгенерировать оглавление. В этом случае не требуется обрабатывать события, возникающие в результате действий пользователя, однако программа все же должна зарегистрировать обработчик события `onload`, чтобы поймать момент, когда закончится загрузка документа и он будет готов к созданию оглавления.

События и обработка событий – это тема главы 17, а данный раздел содержит лишь краткий обзор. События имеют имена, такие как «click», «change», «load», «mouseover», «keypress» или «readystatechange», указывающие общий тип события. События также имеют *адресата* – объект, в котором возникло событие. Ведя речь о событии, необходимо указывать не только его тип (имя), но и адресата, например: событие «click» в объекте `HTMLButtonElement` или событие «readystatechange» в объекте `XMLHttpRequest`.

Если необходимо, чтобы программа откликалась на какое-то событие, необходимо написать функцию, которая называется «обработчиком событий», «приемником событий» или просто «функцией обратного вызова». После этого функцию нужно зарегистрировать, чтобы она вызывалась при появлении события. Как отмечалось выше, это можно сделать с помощью HTML-атрибутов, но такое смешивание JavaScript-кода с разметкой HTML может приводить к путанице. Поэтому обычно лучше регистрировать обработчики событий путем присваивания JavaScript-функций свойствам целевого объекта, как показано ниже:

```
window.onload = function() { ... };
document.getElementById("button1").onclick = function() { ... };
function handleResponse() { ... }
request.onreadystatechange = handleResponse;
```

Обратите внимание, что свойства обработчиков событий имеют имена, которые в соответствии с соглашениями начинаются с префикса «on», за которым следует имя события. Отметьте также, что в примере выше не производится вызов функций: здесь выполняется присваивание функций соответствующим свойствам. Броузер автоматически будет вызывать эти функции при появлении событий. При создании асинхронных обработчиков событий часто используются вложенные функции, и на практике достаточно часто приходится писать программный код, который определяет функции внутри функций, вложенных в другие функции.

В большинстве броузеров для большинства типов событий обработчики получают в виде аргумента объект, свойства которого содержат информацию о событии. Например, объект, передаваемый обработчику события «click», будет иметь свойство, определяющее кнопку мыши, которой был выполнен щелчок. (В IE информация о событии хранится в глобальном объекте и не передается функции-обработчику.) Иногда значение, возвращаемое обработчиком события, используется, чтобы определить, достаточно ли ограничиться выполненной обработкой события и следует ли предотвратить выполнение действий по умолчанию, предусматриваемых броузером.

События, адресатом которых является элемент документа, часто распространяются вверх по дереву документа и этот процесс называется «всплытием». Если, например, пользователь щелкнет мышью на элементе `<button>`, событие «click» будет передано кнопке. Если это событие останется необработанным (и его распространение не будет остановлено) функцией, зарегистрированной в элементе кнопки, событие всплывет до элемента, в который вложена эта кнопка, и будет вызван обработчик события «click», зарегистрированный в этом контейнерном элементе.

Если для одного элемента требуется зарегистрировать более одного обработчика единственного события или если требуется написать модуль, который мог бы безопасно регистрировать обработчики событий, даже если для этих же событий и в этих же объектах уже были зарегистрированы обработчики другим модулем, можно воспользоваться другим приемом регистрации обработчиков. Большинство объектов, которые могут выступать в роли адресата события, имеют метод с именем `addEventListener()`, который позволяет регистрировать множество обработчиков:

```
window.addEventListener("load", function() {...}, false);
request.addEventListener("readystatechange", function() {...}, false);
```

Обратите внимание, что первым аргументом этой функции передается имя события. Несмотря на то что метод `addEventListener()` был определен стандартом уже более десяти лет тому назад, корпорация Microsoft только недавно реализовала его в IE9. В IE8 и в более ранних версиях необходимо использовать похожий метод, который называется `attachEvent()`:

```
window.attachEvent("onload", function() {...});
```

Подробнее о функциях `addEventListener()` и `attachEvent()` рассказывается в главе 17.

Клиентские JavaScript-программы используют еще одну разновидность асинхронных извещений, которые, строго говоря, не являются событиями. Если свойству `onerror` объекта `Window` присвоить функцию, она будет вызываться при появлении ошибочных ситуаций (или необработанных исключений) в программном коде (раздел 14.6). Кроме того, функции `setTimeout()` и `setInterval()` (они являются методами глобального объекта `Window` и потому в клиентском JavaScript считаются глобальными функциями) вызывают указанные им функции по истечении определенного интервала времени. Функции, которые передаются `setTimeout()`, регистрируются не так, как настоящие обработчики событий, и обычно они называются «функциями обратного вызова», а не «обработчиками», но они, как и обработчики событий, выполняются асинхронно. Подробнее о функциях `setTimeout()` и `setInterval()` рассказывается в разделе 14.1.

Пример 13.5 демонстрирует применение функций `setTimeout()`, `addEventListener()` и `attachEvent()` внутри функции `onLoad()`, которая регистрирует обработчик события окончания загрузки документа. `onLoad()` – весьма полезная функция, и мы часто будем использовать ее в примерах на протяжении оставшейся части книги.

Пример 13.5. onLoad(): вызов функции по окончании загрузки документа

```
// Регистрирует функцию f, которая должна вызываться по окончании загрузки документа.
// Если документ уже загружен, функция f будет вызвана асинхронно и в кратчайшие сроки.
function onLoad(f) {
    if (onLoad.loaded)                  // Если документ уже загружен
        window.setTimeout(f, 0);        // Вызвать f, как можно скорее
    else if (window.addEventListener)   // Стандартный метод регистрации событий
        window.addEventListener("load", f, false);
    else if (window.attachEvent)        // В IE8 и в более ранних версиях
        window.attachEvent("onload", f); // используется этот метод
}
// Сначала установить флаг, указывающий, что документ еще не загружен.
onLoad.loaded = false;
// И зарегистрировать функцию, которая сбросит флаг после загрузки документа.
onLoad(function() { onLoad.loaded = true; });
```

13.3.3. Модель потоков выполнения в клиентском JavaScript

Ядро языка JavaScript не имеет механизма одновременного выполнения нескольких потоков управления, и клиентский язык JavaScript не добавляет такой возможности. Стандарт HTML5 определяет механизм поддержки фонового потока выполнения «Web Workers» (подробнее об этом механизме рассказывается ниже), тем не менее JavaScript-код на стороне клиента выполняется в единственном потоке управления. Даже когда параллельное выполнение возможно, интерпретатор JavaScript не может обнаружить этот факт.

Выполнение в единственном потоке существенно упрощает разработку сценариев: можно писать программный код, пребывая в полной уверенности, что два обработчика событий никогда не запустятся одновременно. Можно манипулировать содержимым документа, точно зная, что никакой другой поток выполнения не попытается изменить его в то же самое время, и вам никогда не придется беспокоиться о блокировках, взаимоблокировках или о состояниях гонки за ресурсами при разработке своих программ.

Выполнение в единственном потоке означает, что веб-броузер должен прекратить откликаться на действия пользователя на время выполнения сценария или обработчика события. Это накладывает определенные требования: сценарии и обработчики событий в JavaScript не должны исполняться слишком долго. Если сценарий производит объемные и интенсивные вычисления, это вызовет задержку во время загрузки документа, и пользователь не увидит его содержимое, пока сценарий не закончит свою работу. Если продолжительные по времени операции выполняются в обработчике события, броузер может оказаться неспособным откликаться на действия пользователя, заставляя его думать, что программа «зависла».[1]

Если приложение должно выполнять достаточно сложные вычисления, вызывающие заметные задержки, то перед выполнением таких вычислений следует дать документу возможность полностью загрузиться. Кроме того, полезно предупредить пользователя, что будут производиться длительные вычисления, в процессе которых броузер может не откликаться на его действия. Если есть такая возможность, длительные вычисления следует разбить на несколько подзадач, используя такие методы, как `setTimeout()` и `setInterval()`, для запуска подзадач в фоновом режиме с одновременным обновлением индикатора хода вычислений, предоставляющего обратную связь с пользователем.

Стандарт HTML5 определяет управляемую форму параллельного выполнения – механизм фонового потока выполнения под названием «web worker». Web worker – это фоновый поток выполнения, предназначенный для выполнения продолжительных вычислений и предотвращения блокирования пользовательского интерфейса. Программный код, выполняемый в фоновом потоке web worker, не имеет доступа к содержимому документа, к информации, используемой главным потоком выполнения или другими потоками web worker, и может взаимодействовать с главным потоком и другими фоновыми потоками только посредством асинхронных событий. Благодаря этому параллельное выполнение не оказывает влияния на главный поток, а фоновые потоки не меняют базовую однопоточную модель выполнения JavaScript-программ. Подробнее о фоновых потоках выполнения рассказывается в разделе 22.4.

13.3.4. Последовательность выполнения клиентских сценариев

Мы уже знаем, что выполнение JavaScript-программ начинается с этапа выполнения сценариев и затем переходит к этапу выполнения, управляемому событиями.

---

[1] В некоторых броузерах имеется механизм предотвращения атак типа отказа в обслуживании и случайных зацикливаний, который выводит перед пользователем окно запроса, если сценарий или обработчик события выполняется слишком долго, и позволяет прервать выполнение зациклившегося сценария.

Этот раздел более подробно описывает последовательность выполнения JavaScript-программ.

1. Веб-броузер создает объект `Document` и начинает разбор веб-страницы, добавляя в документ объекты `Element` и текстовые узлы в ходе синтаксического анализа HTML-элементов и их текстового содержимого. На этой стадии свойство `document.readyState` получает значение «loading».
2. Когда механизм синтаксического анализа HTML встречает элементы `<script>`, не имеющие атрибута `async` и/или `defer`, он добавляет эти элементы в документ и затем выполняет встроенные или внешние сценарии. Эти сценарии выполняются синхронно, а на время, пока сценарий загружается (если это необходимо) и выполняется, синтаксический анализ документа приостанавливается. Такие сценарии могут использовать метод `document.write()` для вставки текста во входной поток. Этот текст станет частью документа, когда синтаксический анализ продолжится. Синхронные сценарии часто просто определяют функции и регистрируют обработчики событий для последующего использования, но они могут исследовать и изменять дерево документа, доступное на момент их запуска. То есть синхронные сценарии могут видеть собственный элемент `<script>` и содержимое документа перед ним.
3. Когда механизм синтаксического анализа встречает элемент `<script>`, имеющий атрибут `async`, он начинает загрузку сценария и продолжает разбор документа. Сценарий будет выполнен сразу же по окончании его загрузки, но синтаксический анализ документа не приостанавливается на время загрузки сценария. Асинхронные сценарии не должны использовать метод `document.write()`. Они могут видеть собственный элемент `<script>`, все элементы документа, предшествующие ему и, возможно, дополнительное содержимое документа.
4. По окончании анализа документа значение свойства `document.readyState` изменяется на «interactive».
5. Выполняются все сценарии, имеющие атрибут `defer`, в том порядке, в каком они встречаются в документе. В этот момент также могут выполняться асинхронные сценарии. Отложенные сценарии имеют доступ к полному дереву документа и не должны использовать метод `document.write()`.
6. Броузер возбуждает событие «DOMContentLoaded» в объекте `Document`. Это событие отмечает переход от этапа синхронного выполнения сценариев к управляемому событиями асинхронному этапу выполнения программы. Следует, однако, отметить, в этот период также могут выполняться асинхронные сценарии, которые не были еще выполнены.
7. К этому моменту синтаксический анализ документа завершен, но броузер все еще может ожидать окончания загрузки дополнительного содержимого, такого как изображения. Когда все содержимое будет загружено и все асинхронные сценарии будут выполнены, свойство `document.readyState` получит значение «complete» и веб-броузер возбудит событие «load» в объекте `Window`.
8. С этого момента будут асинхронно вызываться обработчики событий в ответ на действия пользователя, сетевые операции, истечение таймера и т. д.

Это идеализированная последовательность выполнения, и не все броузеры придерживаются ее в точности. Событие «load» поддерживается повсеместно: его возбуждают все броузеры, и оно является универсальным инструментом определения момента окончания загрузки документа и его готовности к выполнению операций. Событие «DOMContentLoaded» возбуждается перед событием «load» и поддерживается всеми текущими броузерами, кроме IE. Свойство `document.readyState` реализовано в большинстве текущих броузеров на момент написания этих строк, но значения, которые получает это свойство, отличаются между броузерами. Атрибут `defer` поддерживается всеми современными версиями IE, но только недавно был реализован в других броузерах. Поддержка атрибута `async` до сих пор не получила широкого распространения, но асинхронное выполнение сценариев с использованием приема, представленного в примере 13.4, поддерживается всеми текущими броузерами. (Однако имейте в виду, что возможность динамической загрузки сценариев с помощью функции, такой как `loadasync()`, размывает границы между этапом загрузки сценариев и этапом выполнения программы, управляемым событиями.)

Данная последовательность не определяет момент, когда документ станет видим пользователю или когда веб-броузер должен начать откликаться на его действия. Все это – особенности каждой конкретной реализации. В случае очень больших документов или очень медленных сетевых подключений теоретически возможно, что веб-броузер будет отображать часть документа и позволять пользователю взаимодействовать с ним до того, как будут выполнены все сценарии. В этом случае действия пользователя могут возбуждать события до начала управляемого событиями этапа выполнения программы.

13.4. Совместимость на стороне клиента

Веб-броузер – это своего рода операционная система для веб-приложений, но Всемирная паутина – это разнородная среда, и ваши веб-документы и приложения будут просматриваться и выполняться в броузерах разных возрастов (от ультрасовременных бета-версий до броузеров, возраст которых исчисляется десятилетиями, таких как IE6), разных производителей (Microsoft, Mozilla, Apple, Google, Opera) и выполняющихся в разных операционных системах (Windows, Mac OS, Linux, iPhone OS, Android). Поэтому достаточно сложно написать нетривиальную клиентскую программу на языке JavaScript, которая будет корректно работать на таком многообразии платформ.

Проблемы совместимости на стороне клиента делятся на три основные категории:

Эволюционные

Веб-платформа постоянно развивается и расширяется. Органы по стандартизации предлагают новые возможности или прикладные интерфейсы. Если какая-то особенность кажется полезной, производители броузеров стремятся реализовать ее. Если достаточно большое число производителей обеспечивают ее совместимость, разработчики начинают ее использовать и опираться на эту особенность, что обеспечивает ей прочное положение в веб-платформе. Иногда инициативу берут на себя производители броузеров и веб-разработчики, и органы по стандартизации включают новую особенность в официальную версию после того, как эта особенность станет стандартом де-факто. В любом случае новая особенность добавляется в Веб. Новые броузеры поддерживают ее, а старые – нет. Веб-разработчики разрываются между желанием использовать новую мощную особенность и стремлением сделать свои веб-страницы пригодными к использованию для как можно более широкого круга посетителей – даже для тех, кто пользуется устаревшими броузерами.

Отсутствие реализации

Иногда мнения производителей броузеров относительно полезности той или иной особенности расходятся. Некоторые производители реализуют ее, другие – нет. Это не проблема отличий между новыми версиями броузеров, поддерживающих особенность, и старыми версиями, не поддерживающими ее. Эта проблема связана с производителями броузеров, одни из которых решили реализовать особенность, а другие – нет. Например, IE8 не поддерживает элемент `<canvas>`, хотя все остальные броузеры реализовали его поддержку. Еще более вопиющий пример: корпорация Microsoft долго отказывалась от намерений реализовать спецификацию DOM Level 2 Events (которая определяет метод `addEventListener()` и связанные с ним). Эта спецификация была выпущена почти десять лет тому назад, и другие производители давно реализовали ее поддержку.[1]

Ошибки

Все броузеры содержат ошибки, и ни один из них не реализует JavaScript API в полном объеме и в точном соответствии со спецификациями. Иногда создание совместимых клиентских сценариев на языке JavaScript является вопросом знания о существовании ошибок в броузерах и умения обходить их.

К счастью, реализации самого языка JavaScript, выполненные различными производителями броузеров, являются совместимыми и не являются источником описываемых проблем. Все броузеры имеют совместимые реализации ECMAScript 3, и к моменту написания этих строк все производители уже работали над реализацией ECMAScript 5. Переход от ECMAScript 3 к ECMAScript 5 также может породить проблемы несовместимости из-за того, что одни броузеры будут поддерживать строгий режим, а другие – нет, но, как ожидается, производители броузеров обеспечат совместимость своих реализаций ECMAScript 5.

При решении проблем несовместимости в клиентских сценариях на языке JavaScript в первую очередь необходимо идентифицировать проблему. Цикл выпуска новых версий веб-броузеров примерно в три раза короче цикла выпуска новых изданий этой книги, поэтому здесь невозможно с достаточной степенью надежности сообщить, какие особенности и в каких версиях броузеров реализованы, тем более невозможно описать ошибки или отличия реализации особенностей в разных броузерах. Такие подробности лучше всего искать в Сети. В процессе работы над стандартом HTML5 рано или поздно должны появиться испытательные тесты. На момент написания этих строк таких тестов еще не существовало, но, как только они появятся, любой желающий сможет получить богатую информацию о совместимости того или иного броузера. А пока я приведу список веб-сайтов, где вы сможете отыскать полезные сведения:

https://developer.mozilla.org

Центр разработчиков Mozilla (Mozilla Developer Center)

---

[1] Справедливости ради следует отметить, что версия IE9 поддерживает и элемент `<canvas>`, и метод `addEventListener()`.

http://msdn.microsoft.com

Сообщество разработчиков, использующих продукты Microsoft (Microsoft Developer Network)

http://developer.apple.com/safari

Центр разработки Safari (Safari Dev Center) на сайте по связям с разработчиками, использующими продукты Apple (Apple Developer Connection)

http://code.google.com/doctype

Компания Google описывает свой проект Doctype как «энциклопедию открытой Сети». Этот сайт, содержимое которого доступно пользователям для редактирования, включает обширные таблицы совместимости для клиентского JavaScript. К моменту написания этих строк данные таблицы содержали только сведения о существовании различных свойств и методов в каждом броузере: они фактически ничего не сообщают о том, насколько корректно работают эти особенности.

http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML_5)[1]

Статья в Википедии, в которой оценивается степень реализации особенностей и API стандарта HTML5 в различных броузерах.

http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(Document_Object_Model)

Аналогичная статья, в которой оценивается степень реализации особенностей DOM.

http://a.deveria.com/caniuse

Сайт «Когда я смогу воспользоваться...» предоставляет информацию о реализации важных веб-особенностей, позволяя фильтровать ее по различным критериям, и дает рекомендации по их использованию, когда останется достаточно мало броузеров, не поддерживающих их.

http://www.quirksmode.org/dom

Содержит таблицы совместимости различных броузеров со спецификациями W3C DOM.

http://webdevout.net/browser-support

Еще один сайт, созданный с целью предоставить информацию о реализации веб-стандартов различными производителями броузеров.

Обратите внимание, что последние три сайта сопровождаются конкретными людьми. Несмотря на то что эти люди являются опытными разработчиками на клиентском JavaScript, эти сайты могут содержать не самую последнюю информацию.

Конечно, понимание проблем несовместимости между броузерами – это только первый шаг. Далее необходимо решить, как обращаться с этими проблемами. Одна из стратегий заключается в том, чтобы ограничиться использованием только тех особенностей, которые одинаково хорошо реализованы (или легко имитируются) во всех броузерах, поддержку которых вам требуется обеспечить. На эту стратегию опирается веб-сайт «Когда я смогу воспользоваться...», упомянутый выше (*http://a.deveria.com/caniuse*): здесь перечислены особенности, ставшие пригодными к широкому использованию после того, как сократилась доля броузера IE6 и он перестал занимать ведущее положение на рынке. В следующих подразделах описывается несколько менее пассивных стратегий, которые можно использовать для обхода несовместимостей на стороне клиента.

---

[1] Похожая статья в Википедии на русском языке: *http://ru.wikipedia.org/wiki/Сравнение_браузеров_(HTML5). – Прим. перев.*

Несколько слов о «текущих броузерах»

Тема клиентского JavaScript изменчива, что стало особенно заметно с появлением ES5 и HTML5. Поскольку платформа развивается очень быстро, я не буду ограничиваться рекомендациями, касающимися конкретных версий тех или иных броузеров: любые такие рекомендации устареют задолго до того, как появится новое издание этой книги. Поэтому вы часто будете видеть, что я специально подстраховываюсь, используя достаточно расплывчатую фразу «все текущие броузеры» (или иногда «все текущие броузеры, кроме IE»). Чтобы добавить конкретики, замечу, что на тот момент, когда я писал эту главу, текущими броузерами (здесь не имеются в виду бета-версии) были:

- Internet Explorer 8
- Firefox 3.6
- Safari 5
- Chrome 5
- Opera 10.10

Когда эта книга поступит в продажу, текущими броузерами, скорее всего, будут: Internet Explorer 9, Firefox 4, Safari 5, Chrome 11 и Opera 11.

Нет никаких гарантий, что каждое сделанное в книге утверждение о «текущих броузерах» в равной степени будет верным для каждого из этих конкретных броузеров. Но вы хотя бы будете знать, какие броузеры считались текущими, когда писалась эта книга.

В пятом издании этой книги вместо выражения «текущие броузеры» использовалось выражение «современные броузеры». То издание было опубликовано в 2006 году, когда текущими броузерами были Firefox 1.5, IE6, Safari 2 и Opera 8.5 (броузер Chrome, созданный компанией Google, тогда еще не существовал). Любые ссылки на «современные броузеры», оставшиеся в этой книге, теперь можно интерпретировать как «все броузеры», потому что версии, более старые, чем эти, практически вышли из употребления.

Многие новейшие особенности клиентского JavaScript, описываемые в этой книге (в частности, в главе 22), реализованы пока не во всех броузерах. Особенности, которые я выбрал для описания в этом издании, – это особенности, для которых процесс стандартизации еще не завершен, но они уже реализованы как минимум в одном выпущенном в свет броузере и находятся в разработке, по крайней мере, еще в одном броузере и, скорее всего, будут приняты всеми производителями броузеров (возможно, за исключением Microsoft).

13.4.1. Библиотеки обеспечения совместимости

Один из самых простых способов избавиться от проблемы несовместимости заключается в использовании библиотек, реализующих обходные решения. Рассмотрим в качестве примера элемент `<canvas>`, предназначенный для создания графических изображений на стороне клиента (эта тема обсуждается в главе 21). Броузер IE является единственным текущим броузером, не поддерживающим эту особенность. Однако он поддерживает собственный, малоизвестный язык создания графических изображений на стороне клиента, который называется VML, с помощью которого можно было бы имитировать действие элемента `<canvas>`. Открытым проектом «explorercanvas» (*http://code.google.com/p/explorercanvas*) была выпущена библиотека, реализующая эту имитацию: достаточно подключить к веб-странице единственный файл *excanvas.js* с программным кодом на языке JavaScript, и броузер IE будет вести себя так, как если бы он поддерживал элемент `<canvas>`.

Библиотека *excanvas.js* может служить ярким примером библиотеки обеспечения совместимости. Точно так же можно написать другие библиотеки, реализующие конкретные особенности. Методы массивов, введенные стандартом ES5 (раздел 7.9), такие как `forEach()`, `map()` и `reduce()`, с успехом можно реализовать в ES3, и добавляя соответствующую библиотеку к страницам, можно получить возможность использовать эти мощные методы как часть базовой платформы любого броузера.

Однако иногда невозможно создать полноценную (или эффективную) реализацию особенности в броузерах, не поддерживающих ее. Как уже упоминалось, IE – единственный броузер, который не реализует стандартный API обработки событий, включая метод `addEventListener()` регистрации обработчиков. Броузер IE поддерживает похожий метод с именем `attachEvent()`. Однако метод `attachEvent()` не такой мощный, как `addEventListener()`, и в действительности не существует очевидного способа реализовать все стандартные методы на основе возможностей, предоставляемых броузером IE. Вместо этого разработчики иногда определяют компромиссный метод обработки событий – часто давая ему имя `addEvent()` – который переносимым способом может использовать либо `addEventListener()`, либо `attachEvent()`. Затем они пишут свой программный код, использующий метод `addEvent()` вместо `addEventListener()` или `attachEvent()`.

На деле многие веб-разработчики в своих веб-страницах используют фреймворки на языке JavaScript, такие как jQuery (описывается в главе 19). Одна из функций, которая делает эти фреймворки такими необходимыми, – определение нового клиентского прикладного интерфейса, совместимого со всеми броузерами. В jQuery, например, регистрация обработчиков событий выполняется с помощью метода `bind()`. Если вы начнете использовать jQuery во всех своих разработках, вам никогда не придется задумываться о несовместимости методов `addEventListener()` и `attachEvent()`. Подробнее о клиентских фреймворках рассказыватся в разделе 13.7.

13.4.2. Классификация броузеров

Классификация броузеров – это прием тестирования и оценки качества, введенный и отстаиваемой компанией Yahoo!, который привносит определенную долю здравомыслия в иначе неуправляемое разрастание вариантов броузеров разных версий от разных производителей и для разных операционных систем. В двух словах, классификация возможностей броузеров подразумевает выделение на основе тестирования броузеров категории «А», которые обеспечивают полную поддержку всех возможностей, и менее мощных броузеров категории «С». Броузеры категории «А» получают полнофункциональные веб-страницы, а броузеры категории «С» – минимальные HTML-версии страниц, в которых не используются сценарии JavaScript и каскадные таблицы стилей CSS. Броузеры, которые не могут быть отнесены к категории «А» или «С», попадают в категорию «Х»: обычно это совершенно новые или особенно редкие броузеры. Считается, что броузеры этой категории обеспечивают полную поддержку всех возможностей, и они получают полнофункциональные веб-страницы, однако официально они не поддерживаются и не тестируются.

Подробности о системе классификации возможностей броузеров, используемой компанией Yahoo!, можно найти на странице *http://developer.yahoo.com/yui/articles/gbs*. На этой же странице приводится текущий список броузеров, включенных компанией Yahoo! в категории «А» и «С» (этот список обновляется ежеквартально). Даже если вы не собираетесь использовать прием классификации броузеров, список броузеров категории «А» может пригодиться для определения, какие броузеры являются текущими и занимают значительную долю рынка.

13.4.3. Проверка особенностей

Проверка особенностей (иногда называемая *проверкой функциональных возможностей*) – это очень мощная методика, позволяющая справиться с проблемами несовместимости. Особенность, или функциональная возможность, которую вы собираетесь использовать, может поддерживаться не всеми броузерами, поэтому необходимо включать в свои сценарии программный код, который будет проверять факт поддержки данной особенности. Если требуемая особенность не поддерживается на текущей платформе, то можно либо не использовать эту особенность на данной платформе, либо разработать альтернативный программный код, одинаково работоспособный на всех платформах.

В следующих главах вы часто будете видеть, что та или иная особенность проверяется снова и снова. Например, в главе 17 приводится программный код, который выглядит, как показано ниже:

```
if (element.addEventListener) { // Проверить наличие метода W3C перед вызовом
    element.addEventListener("keydown", handler, false);
    element.addEventListener("keypress", handler, false);
}
else if (element.attachEvent) { // Проверить наличие метода IE перед вызовом
    element.attachEvent("onkeydown", handler);
    element.attachEvent("onkeypress", handler);
}
else { // В противном случае использовать универсальный прием
    element.onkeydown = element.onkeypress = handler;
}
```

Самое главное, что дает проверка особенностей, – программный код, который не привязан к конкретным броузерам или их версиям. Этот прием работает со всеми броузерами, существующими ныне, и должен продолжить работать с будущими версиями броузеров независимо от того, какой набор особенностей они реализуют. Это означает, что производители броузеров должны определять свойства и методы, обладающие полной функциональностью. Если бы корпорация Microsoft определила метод `addEventListener()`, реализовав спецификации W3C лишь частично, это привело бы к нарушениям работоспособности большого числа сценариев, в которых перед вызовом `addEventListener()` реализован механизм проверки особенностей.

13.4.4. Режим совместимости и стандартный режим

Когда корпорация Microsoft выпустила броузер IE6, в него была добавлена поддержка некоторых стандартных особенностей CSS, которые не поддерживались в IE5. Однако чтобы обеспечить обратную совместимость с существующими веб-страницами, в нем было реализовано два режима отображения. В «стандартном режиме», или в «режиме совместимости с CSS», броузер следует стандартам CSS. В «режиме совместимости» броузер проявляет нестандартное поведение, свойственное версиям IE4 и IE5. Выбор режима отображения зависит от объявления `DOCTYPE` в начале HTML-файла. Страницы, вообще не имеющие объявления `DOCTYPE` и страницы с определенными объявлениями типа документа, типичными в эру использования IE5, отображаются в режиме совместимости. Страницы со строгими объявлениями типа документа (или, для совместимости снизу вверх, с нераспознаваемыми объявлениями типа документа) отображаются в стандартном режиме. Страницы с объявлением, определяемым стандартом HTML5 (`<!DOCTYPE html>`), отображаются в стандартном режиме во всех современных броузерах.

Такое различие между режимом совместимости и стандартным режимом прошло проверку временем. Новые версии IE по-прежнему реализуют его, как и другие современные броузеры, и существование этих двух режимов было узаконено спецификацией HTML5. Различия между режимом совместимости и стандартным режимом обычно имеют значение только для тех, кто пишет HTML- и CSS-код. Однако иногда клиентским сценариям на языке JavaScript бывает необходимо определить, в каком режиме отображается документ. Определить режим отображения можно с помощью свойства `document.compatMode`. Если оно имеет значение «CSS1Compat», документ отображается в стандартном режиме. Если оно имеет значение «BackCompat» (или `undefined`, если такое свойство вообще не существует), документ отображается в режиме совместимости. Все современные броузеры реализуют свойство `compatMode`, и оно стандартизовано спецификацией HTML5.

На практике необходимость проверять свойство `compatMode` возникает редко. Тем не менее в примере 15.8 демонстрируется один из случаев, когда эта проверка действительно необходима.

13.4.5. Проверка типа броузера

Методика проверки особенностей прекрасно подходит для определения поддерживаемых функциональных возможностей броузера. Ее можно использовать, например, чтобы выяснить, какая модель обработки событий поддерживается, W3C или IE. В то же время иногда может потребоваться обойти те или иные ошибки, свойственные конкретному типу броузеров, когда нет достаточно простого способа определить наличие этих ошибок. В этом случае бывает необходимо разработать программный код, который должен выполняться только в броузерах определенного производителя, определенного номера версии или в конкретной операционной системе (либо в некоторой комбинации всех трех признаков).

На стороне клиента сделать это можно с помощью объекта `Navigator`, о котором рассказывается в главе 14. Программный код, который определяет производителя и версию броузера, часто называют *анализатором броузера* (*browser sniffer*) или *анализатором клиента* (*client sniffer*). Простой анализатор такого типа приводится в примере 14.3. Методика определения типа клиента широко использовалась на ранних этапах развития Всемирной паутины, когда Netscape и IE имели серьезные отличия и были несовместимы. Ныне ситуация с совместимостью стабилизировалась, и анализ типа клиента утратил свою актуальность и проводится лишь в тех случаях, когда это действительно необходимо.

Примечательно, что определение типа клиента может быть выполнено также на стороне сервера, благодаря чему веб-сервер на основе строки идентификации броузера, которая передается серверу в заголовке `User-Agent`, может выяснить, какой JavaScript-код требуется отсылать.

13.4.6. Условные комментарии в Internet Explorer

На практике вы можете обнаружить, что большинство несовместимостей, которые необходимо учитывать при разработке клиентских сценариев, обусловлены спецификой броузера IE. Вследствие этого иногда возникает необходимость создавать программный код отдельно для IE и отдельно для всех остальных броузеров. Броузер IE поддерживает нестандартную возможность создания условных комментариев (эта возможность появилась в IE5) в JavaScript-коде, что может оказаться полезным для решения проблем несовместимости.

В следующем примере демонстрируется, как выглядят условные комментарии в HTML. Примечательно, что вся хитрость заключается в комбинации символов, закрывающих комментарий.

```
<!--[if IE 6]>
Эти строки фактически находятся внутри HTML-комментария.
Они будут отображаться только в IE6.
<![endif]-->

<!--[if lte IE 7]>
Эта строка будет отображена только в IE 5, 6, 7 и в более ранних версиях.
lte обозначает "less than or equal" (меньше или равно). Можно также использовать "lt",
"gt" и "gte".
<![endif]-->

<!--[if !IE]> <-->
Это обычное HTML-содержимое, но IE не будет отображать его
из-за комментариев, что расположены выше и ниже.
<!--> <![endif]-->

Это обычное содержимое, которое будет отображаться всеми броузерами.
```

В качестве более конкретного примера возьмем библиотеку *excanvas.js*, о которой выше говорилось, что она реализует поддержку элемента `<canvas>` в Internet Explorer. Поскольку эта библиотека требуется, только когда веб-страница отображается в IE (и работает только в IE), есть смысл оформлять ее подключение внутри условного комментария, чтобы другие броузеры даже не загружали ее:

```
<!--[if IE]><script src="excanvas.js"></script><![endif]-->
```

Условные комментарии также поддерживаются интерпретатором JavaScript в IE. Программисты, знакомые с языком C/C++, найдут их похожими на инструкции препроцессора `#ifdef`/`#endif`. Условные JavaScript-комментарии в IE начинаются с комбинации символов `/*@cc_on` и завершаются комбинацией `@*/`. (Префиксы «cc» и «cc_on» происходят от фразы «condition compilation», т. е. «условная компиляция».) Следующий условный комментарий содержит программный код, который будет выполняться только в IE:

```
/*@cc_on
  @if (@_jscript)
    // Следующий код находится внутри JS-комментария, но IE выполнит его.
    alert("In IE");
  @end
  @*/
```

Внутри условных комментариев могут указываться ключевые слова `@if`, `@else` и `@end`, предназначенные для отделения программного кода, который должен выполняться интерпретатором JavaScript в IE по определенному условию. В большинстве случаев вам достаточно будет использовать показанное в предыдущем фрагменте условие `@if (@_jscript)`. JScript – это название интерпретатора JavaScript, которое было дано ему в Microsoft, а переменная `@_jscript` в IE всегда имеет значение `true`.

При грамотном чередовании условных и обычных JavaScript-комментариев можно определить, какой блок программного кода должен выполняться в IE, а какой – во всех остальных броузерах:

```
/*@cc_on
  @if (@_jscript)
    // Этот блок кода находится внутри условного комментария,
    // который также является обычным JavaScript-комментарием.
    // В IE этот блок будет выполнен, а в других броузерах - нет.
    alert('Вы пользуетесь Internet Explorer');
  @else*/
    // Этот блок уже не находится внутри JavaScript-комментария,
    // но по-прежнему находится внутри условного комментария IE.
    // Вследствие этого данный блок кода будет выполнен всеми броузерами, кроме IE.
    alert('Вы не пользуетесь Internet Explorer');
  /*@end
@*/
```

13.5. Доступность

Всемирная паутина представляет собой замечательный инструмент распространения информации, и JavaScript-сценарии могут сделать эту информацию максимально доступной. Однако JavaScript-программисты должны проявлять осторожность: слишком просто написать такой JavaScript-код, который сделает невозможным восприятие информации для пользователей с ограниченными возможностями.

Пользователи с ослабленным зрением применяют такие «вспомогательные технологии», как программы чтения с экрана, когда слова, выводимые на экран, преобразуются в речевые аналоги. Некоторые программы чтения с экрана способны распознавать JavaScript-код, другие лучше работают, когда поддержка JavaScript отключена. Если вы разрабатываете сайт, который требует выполнения JavaScript-кода на стороне клиента для отображения информации, вы ограничиваете доступность своего сайта для пользователей подобных программ чтения с экрана. (И ограничиваете тех, кто преднамеренно отключил поддержку JavaScript в броузере.) Главная цель JavaScript заключается в улучшении представления информации, а не собственно в ее представлении. Основное правило применения JavaScript заключается в том, что веб-страница, в которую встроен JavaScript-код, должна оставаться работоспособной (хотя бы ограниченно), даже когда интерпретатор JavaScript отключен.

Другое важное замечание относительно доступности касается пользователей, которые могут работать с клавиатурой, но не могут (или не хотят) применять указывающие устройства, такие как мышь. Если программный код ориентирован на события, возникающие от действий мышью, вы ограничиваете доступность страницы для тех, кто не пользуется мышью. Веб-броузеры позволяют задействовать клавиатуру для перемещения по веб-странице и выполнять операции с элементами графического интерфейса в них, то же самое должен позволять делать и ваш JavaScript-код. Как демонстрируется в главе 17, наряду с поддержкой событий, зависящих от типа устройства, таких как `onmouseover` или `onmousedown`, JavaScript обладает поддержкой событий, не зависящих от типа устройства, таких как `onfocus` и `onchange`. Для достижения максимальной доступности следует отдавать предпочтение событиям, не зависящим от типа устройства.

Создание максимально доступных веб-страниц – нетривиальная задача, а обсуждение проблем обеспечения доступности выходит за рамки этой книги. Разработчики веб-приложений, для которых проблемы доступности имеют немаловажное значение, должны ознакомиться со стандартами WAI-ARIA (Web Accessibility Initiative – Accessible Rich Internet Applications) *http://www.w3.org/WAI/intro/aria*.

13.6. Безопасность

Наличие интерпретаторов JavaScript в веб-броузерах означает, что загружаемая веб-страница может вызвать на выполнение произвольный JavaScript-код. Это требует обеспечения надлежащей безопасности, и производители броузеров прилагают немалые усилия для достижения двух противоречивых целей:

- Реализовать мощный прикладной программный интерфейс на стороне клиента, позволяющий писать полезные веб-приложения.
- Воспрепятствовать злонамеренному программному коду читать или изменять персональные данные пользователя, получать доступ к конфиденциальным данным, обманывать пользователя или тратить его время.

Как и во многих других областях, обеспечение безопасности в JavaScript представляет собой непрерывный процесс обнаружения и решения проблем. В самом начале развития Всемирной паутины в броузеры была добавлена возможность открывать и перемещать окна, изменять их размеры и выводить произвольный текст в строке состояния броузера. Когда недобросовестные рекламодатели и жулики начали злоупотреблять этой возможностью, производители броузеров начали ограничивать или вообще запрещать использование соответствующих функций. В настоящее время, работая над стандартом HTML5, производители броузеров с осторожностью (открыто и совместно) отменяют некоторые давнишние ограничения, связанные с безопасностью и расширяют возможности клиентского JavaScript, не добавляя (хотелось бы надеяться) новых дыр в системе безопасности.

В следующих подразделах рассматриваются ограничения JavaScript и проблемы безопасности, с которыми вы, как веб-разработчик, должны быть знакомы.

13.6.1. Чего не может JavaScript

Броузеры – это первая линия обороны против злонамеренного кода, поэтому они просто не поддерживают некоторые функциональные возможности. Например, клиентский JavaScript не предоставляет никакого способа записи или удаления файлов и каталогов на клиентском компьютере. То есть программа на языке JavaScript не может удалить данные или установить вирус. (Тем не менее в разделе 22.6.5 вы узнаете, как на языке JavaScript реализовать чтение файлов, выбранных пользователем, а в разделе 22.7 – как настроить безопасную частную файловую систему, в пределах которой программы на языке JavaScript смогут работать с файлами.)

Аналогично клиентский JavaScript не имеет универсальных механизмов сетевых взаимодействий. Клиентский сценарий на языке JavaScript может управлять протоколом HTTP (как описывается в главе 18). А другой стандарт, примыкающий к стандарту HTML5, известный как WebSockets, определяет прикладной программный интерфейс, напоминающий сокеты, позволяющий взаимодействовать со специализированными серверами. Но ни один из этих интерфейсов не позволяет получить непосредственный доступ к сети. На клиентском JavaScript нельзя написать программу универсального сетевого клиента или сервера.

Вторая линия обороны против злонамеренного программного кода – это наложение ограничений на некоторые поддерживаемые функциональные возможности. Ниже перечислены некоторые ограничения:

- JavaScript-программа может открыть новое окно броузера, но из-за того, что многие рекламодатели злоупотребляют этой возможностью, большинство броузеров позволяют ограничить эту возможность так, чтобы всплывающие окна могли появиться только в ответ на действия пользователя, такие как щелчок мыши.
- JavaScript-программа может закрыть окно броузера, открытое ею же, но она не может закрыть другое окно без подтверждения пользователя.
- Свойство `value` HTML-элемента `FileUpload` не может быть установлено программно. Если бы это свойство было доступно, сценарий мог бы установить его значение равным любому желаемому имени файла и заставить форму выгрузить на сервер содержимое любого указанного файла (например, файла паролей).
- Сценарий не может прочитать содержимое документов с других серверов, отличных от сервера, откуда был получен документ с данным сценарием. Аналогичным образом сценарий не может зарегистрировать обработчики событий в документах, полученных с других серверов. Это предотвращает возможность подсматривания данных, вводимых пользователем (таких как комбинации символов, составляющих пароль) в других страницах. Это ограничение известно как *политика общего происхождения* (*same-origin policy*) и более подробно описывается в следующем разделе.

Обратите внимание, что это далеко не полный список ограничений, имеющихся в клиентском JavaScript. Различные броузеры используют различные стратегии безопасности и могут накладывать различные ограничения. Кроме того, некоторые броузеры могут также предоставлять возможность ужесточать или ослаблять ограничения с помощью пользовательских настроек.

13.6.2. Политика общего происхождения

Политикой общего происхождения называется радикальное ограничение, связанное с безопасностью, накладываемое на веб-содержимое, с которым может взаимодействовать JavaScript-код. Обычно политика общего происхождения вступает в игру, когда веб-страница содержит элементы `<iframe>` или открывает другие окна броузера. В этом случае политика общего происхождения ограничивает возможность JavaScript-кода в одном окне взаимодействовать с содержимым в других окнах и фреймах. В частности, сценарий может читать только свойства окон и документов, имеющих общее с самим сценарием происхождение (о том, как использовать JavaScript для работы с несколькими окнами и фреймами, рассказывается в разделе 14.8).

Происхождение документа определяется протоколом, именем хоста и номером порта URL-адреса, откуда был загружен документ. Документы, загружаемые с других веб-серверов, имеют другое происхождение. Документы, загруженные с разных портов одного и того же хоста, также имеют другое происхождение. Наконец, документы, загруженные по протоколу `http:`, по происхождению отличаются от документов, загруженных по протоколу `https:`, даже если загружены с одного и того же веб-сервера.

Важно понимать, что происхождение самого сценария не имеет никакого отношения к политике общего происхождения: значение имеет происхождение документа, в который встраивается сценарий. Предположим, что сценарий, хранящийся на сервере A, включается (с помощью атрибута `src` элемента `<script>`) в веб-страницу, обслуживаемую сервером B. С точки зрения политики общего происхождения будет считаться, что этот сценарий происходит с сервера B и он получит иметь полный доступ ко всему содержимому этого документа. Если этот сценарий откроет второе окно и загрузит в него документ с сервера B, он также будет иметь полный доступ к содержимому этого второго документа. Но если сценарий откроет третье окно и загрузит в него документ с сервера C (или даже с сервера A), в дело вступит политика общего происхождения и ограничит сценарий в доступе к этому документу.

Политика общего происхождения на самом деле применяется не ко всем свойствам всех объектов в окне, имеющем другое происхождение, но она применяется ко многим из них, в частности, практически ко всем свойствам объекта `Document`. В любом случае можно считать, что любое окно или фрейм, содержащий документ, полученный с другого сервера, для ваших сценариев будут закрыты. Если такое окно было открыто самим сценарием, он сможет закрыть его, но не может «заглянуть внутрь» окна. Кроме того, политика общего происхождения действует при работе по протоколу HTTP с применением объекта `XMLHttpRequest` (глава 18). Этот объект позволяет JavaScript-сценариям, выполняющимся на стороне клиента, отправлять произвольные HTTP-запросы, но только тому веб-серверу, откуда был загружен документ, содержащий сценарий.

Политика общего происхождения необходима, чтобы не допустить хищение конфиденциальной информации. Без этого ограничения злонамеренный сценарий (возможно, загруженный в броузер, расположенный в защищенной брандмауэром корпоративной сети) мог бы открыть пустое окно, в надежде обмануть пользователя и заставить его задействовать это окно для поиска файлов в локальной сети. После этого злонамеренный сценарий мог бы прочитать содержимое этого окна и отправить его обратно на свой сервер. Политика общего происхождения предотвращает возможность возникновения такого рода ситуаций.

13.6.2.1. Ослабление ограничений политики общего происхождения

В некоторых ситуациях политика общего происхождения оказывается слишком строгой. Поэтому в этом разделе описывается, как можно ослабить накладываемые ею ограничения.

Политика общего происхождения создает определенные проблемы для крупных веб-сайтов, которые функционируют на нескольких серверах. Например, сценарий с сервера *home.example.com* мог бы на вполне законных основаниях читать свойства документа, загруженного с *developer.example.com*, а сценариям с *orders.example.com* может потребоваться прочитать свойства из документов с *catalog.example.com*. Для поддержки таких крупных веб-сайтов можно использовать свойство `domain` объекта `Document`. По умолчанию свойство `domain` содержит имя сервера, с которого был загружен документ. Это свойство можно установить только равным строке, являющейся допустимым доменным суффиксом первоначального значения. Другими словами, если значение свойства `domain` первоначально было равно строке «home.example.com», то можно установить его равным «example.com», но не «home.example» или «ample.com». Кроме того, значение свойства `domain` должно содержать, по крайней мере, одну точку, чтобы его нельзя было установить равным «com» или другому имени домена верхнего уровня.

Если два окна (или фрейма) содержат сценарии, установившие одинаковые значения свойства `domain`, политика общего происхождения для этих двух окон ослабляется, и каждое из окон может читать значения свойств другого окна. Например, взаимодействующие сценарии в документах, загруженных с серверов *orders.example.com* и *catalog.example.com*, могут установить свойства `document.domain` равными «example.com», тем самым указывая на общность происхождения документов и разрешая каждому из документов читать свойства другого.

Второй прием ослабления ограничений политики общего происхождения предполагается стандартизовать под названием «Cross-Origin Resource Sharing» (*http://www.w3.org/TR/cors/*). Этот проект стандарта дополняет протокол HTTP новым заголовком запроса `Origin:` и новым заголовком ответа `Access-Control-Allow-Origin`. Он позволяет серверам использовать заголовок для явного определения списка доменов, которые могут запрашивать файл, или использовать шаблонные символы, чтобы обеспечить возможность получения файла любым сайтом. Броузеры, такие как Firefox 3.5 и Safari 4, используют этот новый заголовок, чтобы разрешить выполнение междоменных HTTP-запросов с использованием объекта `XMLHttpRequest`, которые иначе были бы невозможны из-за ограничений политики общего происхождения.

Еще один новый прием, известный как «обмен сообщениями между документами» (cross-document messaging), позволяет сценарию из одного документа передавать текстовые сообщения сценарию в другом документе независимо от домена происхождения сценария. Вызов метода `postMessage()` объекта `Window` производит асинхронную отправку сообщения (получить которое можно в обработчике события `onmessage`) документу в этом окне. Сценарий в одном документе по-прежнему лишен возможности вызывать методы или читать свойства другого документа, но они могут безопасно взаимодействовать друг с другом, используя прием обмена сообщениями. Подробнее API обмена сообщениями между документами рассматривается в разделе 22.3.

13.6.3. Взаимодействие с модулями расширения и элементами управления ActiveX

Хотя в базовом языке JavaScript и базовой объектной модели на стороне клиента отсутствуют средства для работы с сетевым окружением и файловой системой, которые необходимы наихудшему злонамеренному программному коду, тем не менее ситуация не такая простая, как кажется на первый взгляд. Во многих броузерах JavaScript-код используется как «механизм выполнения» других программных компонентов, таких как элементы управления ActiveX (в IE) и модули расширения (в других броузерах). Самыми распространенными примерами являются модули расширения, обеспечивающие поддержку Flash и Java, и они предоставляют в распоряжение клиентских сценариев дополнительные мощные возможности.

Вопросы безопасности приобретают особую важность, когда речь заходит о передаче управления элементам ActiveX и модулям расширения. Апплеты Java, например, могут иметь низкоуровневый доступ к сетевым возможностям. Защитная «песочница» для Java не позволяет апплетам взаимодействовать с серверами, отличными от того, откуда они были получены; тем самым закрывается брешь в системе безопасности. Но остается основная проблема: если модуль расширения может управляться из сценария, необходимо полное доверие не только системе безопасности броузера, но и системе безопасности самого модуля расширения. На практике модули расширения Java и Flash, похоже, не имеют проблем с безопасностью и не вызывают появление этих проблем в клиентских сценариях на языке JavaScript. Однако элементы управления ActiveX имеют более пестрое прошлое. Броузер IE обладает возможностью доступа из сценариев к самым разным элементам управления ActiveX, которые являются частью операционной системы Windows и которые раньше уже были источниками проблем безопасности.

13.6.4. Межсайтовый скриптинг

Термин *межсайтовый скриптинг* (cross-site scripting), или XSS, относится к области компьютерной уязвимости, когда атакующий внедряет HTML-теги или сценарии в документы на уязвимом веб-сайте. Организация защиты от XSS-атак – обычное дело для веб-разработчиков, занимающихся созданием серверных сценариев. Однако программисты, разрабатывающие клиентские JavaScript-сценарии, также должны знать о XSS-атаках и предпринимать меры защиты от них.

Веб-страница считается уязвимой для XSS-атак, если она динамически создает содержимое документа на основе пользовательских данных, не прошедших предварительную обработку по удалению встроенного HTML-кода. В качестве тривиального примера рассмотрим следующую веб-страницу, которая использует JavaScript-сценарий, чтобы приветствовать пользователя по имени:

```
<script>
var name = decodeURIComponent(window.location.search.substring(1)) || "";
document.write("Привет, " + name);
</script>
```

В этом двустрочном сценарии используется метод `window.location.search.substring`, с помощью которого извлекается часть адресной строки, начинающаяся с символа ?. Затем с помощью метода `document.write()` добавляется динамически сгенерированное содержимое документа. Этот сценарий предполагает, что обращение к веб-странице будет производиться с помощью следующего URL-адреса:

```
http://www.example.com/greet.html?name=Давид
```

В этом случае будет выведен текст «Привет, Давид». Но что произойдет, если страница будет запрошена с использованием следующего URL-адреса:

```
http://www.example.com/greet.html?name=%3Cscript%3Ealert('Давид')%3C/script%3E
```

С таким содержимым URL-адреса сценарий динамически сгенерирует другой сценарий (коды `%3C` и `%3E` – это угловые скобки)! В данном случае вставленный сценарий просто отобразит диалоговое окно, которое не представляет никакой опасности. Но представьте себе такой случай:

```
http://siteA/greet.html?name=%3Cscript src=siteB/evil.js%3E%3C/script%3E
```

Межсайтовый скриптинг потому так и называется, что в атаке участвует более одного сайта. Сайт B (или даже сайт C) включает специально сконструированную ссылку (подобную только что показанной) на сайт A, в которой содержится сценарий с сайта B. Сценарий *evil.js* размещается на сайте злоумышленника B, но теперь этот сценарий оказывается внедренным в сайт A и может делать все, что ему заблагорассудится с содержимым сайта A. Он может стереть страницу или вызвать другие нарушения в работе сайта (такие как отказ в обслуживании, о чем рассказывается в следующем разделе). Это может отрицательно сказаться на посетителях сайта A. Гораздо опаснее, что такой злонамеренный сценарий может прочитать содержимое cookies, хранящихся на сайте A (возможно, содержащих учетные номера или другие персональные сведения), и отправить эти данные обратно на сайт B. Внедренный сценарий может даже отслеживать нажатия клавиш и отправлять эти данные на сайт B.

Универсальный способ предотвращения XSS-атак заключается в удалении HTML-тегов из всех данных сомнительного происхождения, прежде чем использовать их для динамического создания содержимого документа. Чтобы исправить эту проблему в показанном ранее файле *greet.html*, нужно добавить следующую строку в сценарий, которая призвана удалять угловые скобки, окружающие тег `<script>`:

```
name = name.replace(/</g, "&lt;").replace(/>/g, "&gt;");
```

Простейшая операция, представленная выше, заменит в строке все угловые скобки на соответствующие им мнемоники языка HTML, тем самым экранировав и деактивировав любые HTML-теги, присутствующие в строке. IE8 определяет более точный метод `toStaticHTML()`, который удаляет все теги `<script>` (и любое другое выполняемое содержимое) без изменения невыполняемой разметки HTML.

Метод `toStaticHTML()` не является стандартным, но для вас не составит труда самостоятельно написать аналогичную функцию экранирования разметки HTML на базовом JavaScript.

Стандарт HTML5 пошел еще дальше по пути развития стратегий защиты и определяет атрибут `sandbox` в элементе `<iframe>`. Когда этот атрибут будет реализован, он должен будет обеспечивать безопасное отображение непроверенного содержимого, автоматически запрещая выполнения сценариев, имеющихся в нем.

Межсайтовый скриптинг представляет собой уязвимость, глубоко уходящую корнями в архитектуру Всемирной паутины. Вы должны осознавать всю глубину этой уязвимости, но дальнейшее ее обсуждение выходит далеко за рамки темы данной книги. В Интернете есть немало ресурсов, которые помогут вам организовать защиту от атак подобного рода. Наиболее важный из них принадлежит группе компьютерной «скорой помощи» CERT Advisory: *http://www.cert.org/advisories/CA-2000-02.html*.

13.6.5. Атаки типа отказа в обслуживании

Политика общего происхождения и другие меры безопасности прекрасно защищают данные клиента от посягательств со стороны злонамеренного программного кода, но не могут предотвратить атак типа отказа в обслуживании. При посещении злонамеренного веб-сайта ваш броузер может получить JavaScript-сценарий, который в бесконечном цикле вызывает метод `alert()`, выводящий диалоговое окно или нагружающий центральный процессор компьютера бесконечным циклом с бессмысленными вычислениями.

Некоторые типы броузеров автоматически определяют наличие циклов, выводящих диалоговые окна или имеющих продолжительное время работы, и предоставляют пользователю возможность прервать их. Но злонамеренный программный код для обхода этой защиты может использовать прием на основе метода `setInterval()`, загружая процессор, а также атаковать систему клиента, вызывая огромный расход памяти. В веб-броузерах не существует универсального способа предотвращения таких атак. На самом деле это не является распространенной проблемой Всемирной паутины, т. к. никто второй раз не посетит злонамеренный сайт!

13.7. Клиентские фреймворки

Многие веб-разработчики считают удобным конструировать свои веб-приложения на основе библиотек клиентских фреймворков. Эти библиотеки являются «основой» в том смысле, что они создают новый, высокоуровневый прикладной интерфейс для клиентских сценариев, действующий поверх стандартного и собственного API веб-броузеров: приняв за основу какой-либо фреймворк, вам остается только использовать определяемый им прикладной интерфейс в своем программном коде. Одним из очевидных преимуществ использования фреймворков является высокоуровневый прикладной интерфейс, позволяющий выполнять больше операций за счет меньшего объема программного кода. Кроме того, добротно выполненные фреймворки решают многие проблемы совместимости, безопасности и доступности, описанные выше.

В главе 19 мы познакомимся с jQuery – одним из наиболее популярных фреймворков. Если вы решите использовать jQuery в своих проектах, вам все равно желательно прочитать главы, предшествующие главе 19, – понимание низкоуровневых прикладных интерфейсов повысит ваш профессиональный уровень, даже если вам редко придется использовать эти интерфейсы.

Помимо jQuery существует еще множество фреймворков на языке JavaScript – их слишком много, чтобы перечислять здесь. Тем не менее ниже я перечислил некоторые из наиболее известных и широко используемых фреймворков:

Prototype

Как и jQuery, библиотека Prototype (*http://prototypejs.org*) основное внимание уделяет работе с DOM и Ajax и добавляет довольно много вспомогательных функций в базовый язык. В дополнение к ней можно добавить библиотеку Scriptaculous (*http://script.aculo.us/*), которая реализует анимационные и визуальные эффекты.

Dojo

Dojo (*http://dojotoolkit.org*) – это огромный фреймворк, имеющий «невероятную глубину». Он включает обширный набор графических элементов пользовательского интерфейса, системный пакет, уровень абстракции данных и многое другое.

YUI

YUI (*http://developer.yahoo.com/yui/*) – это библиотека, созданная компанией Yahoo! для внутренних нужд и используемая ею в своих веб-страницах. Подобно Dojo, это большая, всеохватывающая библиотека утилит и функций для работы с DOM, виджетами и т. д. Существует две несовместимые версии, известные как YUI 2 и YUI 3.

Closure

Библиотека Closure (*http://code.google.com/closure/library/*) – это клиентская библиотека, которую компания Google использует в своих веб-приложениях Gmail, Google Docs и в других. Эта библиотека предназначена для использования совместно с компилятором Closure (*http://code.google.com/closure/compiler/*), который удаляет из библиотеки неиспользуемые функции. Благодаря тому, что перед развертыванием веб-приложения из него удаляется весь ненужный программный код, создатели библиотеки Closure избежали необходимости заботиться о ее компактности и добавили в библиотеку Closure обширный набор утилит.

GWT

GWT, Google Web Toolkit (*http://code.google.com/webtoolkit/*), – это совершенно иной тип клиентских фреймворков. Он определяет прикладной интерфейс веб-приложений на языке Java и предоставляет компилятор для преобразования Java-программ в совместимые сценарии на клиентском JavaScript. Фреймворк GWT используется в некоторых программных продуктах компании Google, но не так широко, как библиотека Closure.

14

Объект Window

В главе 13 был представлен объект `Window` и отмечена центральная роль, которую этот объект играет в клиентском JavaScript: объект `Window` является глобальным объектом для клиентских JavaScript-программ. В этой главе будут рассмотрены свойства и методы объекта `Window`. Эти свойства определяют множество различных API, из которых лишь немногие имеют отношение к окну броузера, в честь которого этот объект получил имя `Window`. В этой главе рассматриваются следующие темы:

- В разделе 14.1 демонстрируется, как с помощью функций `setTimeout()` и `setInterval()` зарегистрировать собственную функцию для вызова в определенные моменты времени в будущем.
- В разделе 14.2 описывается, как использовать свойство `location`, чтобы получить URL-адрес текущего документа и загрузить новый документ.
- В разделе 14.3 описывается свойство `history` и демонстрируется, как перемещаться взад и вперед по списку ранее посещавшихся страниц.
- В разделе 14.4 демонстрируется, как использовать свойство `navigator` для получения информации о производителе броузера и как использовать свойство `screen` для определения размеров рабочего стола.
- В разделе 14.5 демонстрируется, как выводить простые диалоги с текстовыми сообщениями, используя методы `alert()`, `confirm()` и `prompt()`, и как отображать диалоги с разметкой HTML с помощью метода `showModalDialog()`.
- В разделе 14.6 описывается, как регистрировать обработчик `onerror`, который будет вызываться при появлении необработанных исключений.
- В разделе 14.7 рассказывается о том, что идентификаторы и имена HTML-элементов используются в качестве свойств объекта `Window`.
- В самом длинном разделе 14.8 описывается, как открывать и закрывать окна броузера и как писать JavaScript-код, призванный взаимодействовать с несколькими окнами и фреймами.

14.1. Таймеры

Функции `setTimeout()` и `setInterval()` позволяют зарегистрировать функцию, которая будет вызываться один или более раз через определенные интервалы времени. Это очень важные функции для клиентского JavaScript, и поэтому они были определены как методы объекта `Window`, несмотря на то что являются универсальными функциями, не выполняющими никаких действий с окном.

Метод `setTimeout()` объекта `Window` планирует запуск функции через определенное число миллисекунд. Метод `setTimeout()` возвращает значение, которое может быть передано методу `clearTimeout()`, чтобы отменить запланированный ранее запуск функции.

Метод `setInterval()` похож на `setTimeout()`, за исключением того, что он автоматически заново планирует повторное выполнение через указанное количество миллисекунд:

```
setInterval(updateClock, 60000); // Вызывать updateClock() через каждые 60 сек.
```

Подобно `setTimeout()`, метод `setInterval()` возвращает значение, которое может быть передано методу `clearInterval()`, чтобы отменить запланированный запуск функции.

В примере 14.1 определяется вспомогательная функция, которая ожидает указанный интервал времени, многократно вызывает указанную функцию и затем отменяет запланированные вызовы по истечении другого заданного интервала времени. Этот пример демонстрирует использование методов `setTimeout()`, `setInterval()` и `clearInterval()`.

Пример 14.1. Вспомогательная функция для работы с таймером

```
/*
 * Планирует вызов или вызовы функции f() в будущем.
 * Ожидает перед первым вызовом start миллисекунд, затем вызывает f()
 * каждые interval миллисекунд и останавливается через start+end миллисекунд.
 * Если аргумент interval указан, а аргумент end нет, повторяющиеся вызовы функции f
 * никогда не прекратятся. Если отсутствуют оба аргумента, interval и end,
 * тогда функция f будет вызвана только один раз, через start миллисекунд.
 * Если указан только аргумент f, функция будет вызвана немедленно, как если бы
 * в аргументе start было передано число 0. Обратите внимание, что вызов invoke()
 * не блокируется: она сразу же возвращает управление.
 */
function invoke(f, start, interval, end) {
    if (!start) start = 0;        // По умолчанию через 0 мс
    if (arguments.length <= 2)    // Случай однократного вызова
        setTimeout(f, start);     // Вызвать 1 раз через start мс.
    else {                        // Случай многократного вызова
        setTimeout(repeat, start); // Начать вызовы через start мс
        function repeat() {       // Планируется на вызов выше
            var h = setInterval(f, interval); // Вызывать f через interval мс.
            // Прекратить вызовы через end мс, если значение end определено
            if (end) setTimeout(function() { clearInterval(h); }, end);
        }
    }
}
```

По исторически сложившимся причинам в первом аргументе методам `setTimeout()` и `setInterval()` допускается передавать строку. В этом случае строка будет интерпретироваться (как с применением функции `eval()`) через указанный интервал времени. Спецификация HTML5 (и все броузеры, кроме IE) допускает передавать методам `setTimeout()` и `setInterval()` дополнительные аргументы после первых двух. Все эти дополнительные аргументы будут передаваться функции, вызов которой планируется этими методами. Однако если требуется сохранить совместимость с IE, эту возможность использовать не следует.

Если методу `setTimeout()` указать величину интервала 0 миллисекунд, указанная функция будет вызвана не сразу, а «как только такая возможность появится», т. е. как только завершат работу все обработчики событий.

14.2. Адрес документа и навигация по нему

Свойство `location` объекта `Window` ссылается на объект `Location`, представляющий текущий URL-адрес документа, отображаемого в окне и определяющий методы, инициирующие загрузку нового документа в окно.

Свойство `location` объекта `Document` также ссылается на объект `Location`:

```
window.location === document.location // всегда верно
```

Кроме того, объект `Document` имеет свойство `URL`, хранящее статическую строку с адресом URL документа. При перемещении по документу с использованием идентификаторов фрагментов (таких как «#table-of-contents») внутри документа объект `Location` будет обновляться, отражая факт перемещения, но свойство `document.URL` останется неизменным.

14.2.1. Анализ URL

Свойство `location` окна является ссылкой на объект `Location` и представляет URL-адрес документа, отображаемого в данный момент в текущем окне. Свойство `href` объекта `Location` – это строка, содержащая полный текст URL-адреса. Метод `toString()` объекта `Location` возвращает значение свойства `href`, поэтому в контекстах, где неявно подразумевается вызов метода `toString()`, вместо конструкции `location.href` можно писать просто `location`.

Другие свойства этого объекта, такие как `protocol`, `host`, `hostname`, `port`, `pathname`, `search` и `hash`, определяют отдельные части URL-адреса. Они известны как свойства «декомпозиции URL» и также поддерживаются объектами `Link` (которые создаются элементами `<a>` и `<area>` в HTML-документах). Полное описание объектов `Location` и `Link` приводится в четвертой части книги.

Свойства `hash` и `search` объекта `Location` представляют особый интерес. Свойство `hash` возвращает «идентификатор фрагмента» из адреса URL, если он имеется: символ решетки (#) со следующим за ним идентификатором. Свойство `search` содержит часть URL-адреса, следующую за вопросительным знаком, если таковая имеется, включая сам знак вопроса. Обычно эта часть URL-адреса является строкой запроса. В целом эта часть URL-адреса используется для передачи параметров и является средством встраивания аргументов в URL-адрес. Хотя эти аргументы обычно предназначены для сценариев, выполняющихся на сервере, нет

никаких причин, по которым они не могли бы также использоваться в страницах, содержащих JavaScript-код. В примере 14.2 приводится определение универсальной функции `urlArgs()`, позволяющей извлекать аргументы из свойства `search` URL-адреса. В примере используется глобальная функция `decodeURIComponent()`, имеющаяся в клиентском JavaScript. (Смотрите справочную статью «Global» в третьей части книги.)

Пример 14.2. Извлечение аргументов из строки search URL-адреса

```
/*
 * Эта функция выделяет в URL-адресе разделенные амперсандами
 * пары аргументов имя/значение из строки запроса. Сохраняет эти пары
 * в свойствах объекта и возвращает этот объект. Порядок использования:
 *
 * var args = urlArgs(); // Извлечь аргументы из URL
 * var q = args.q || ""; // Использовать аргумент, если определен,
                         // или значение по умолчанию
 * var n = args.n ? parseInt(args.n) : 10;
 */
function urlArgs() {
    var args = {};                               // Создать пустой объект
    var query = location.search.substring(1);    // Строка запроса без '?'
    var pairs = query.split("&");                // Разбить по амперсандам
    for(var i = 0; i < pairs.length; i++) {      // Для каждого фрагмента
        var pos = pairs[i].indexOf('=');         // Отыскать пару имя/значение
        if (pos == -1) continue;                 // Не найдено - пропустить
        var name = pairs[i].substring(0,pos);    // Извлечь имя
        var value = pairs[i].substring(pos+1);   // Извлечь значение
        value = decodeURIComponent(value);       // Преобразовать значение
        args[name] = value;                      // Сохранить в виде свойства
    }
    return args;                                 // Вернуть полученные аргументы
}
```

14.2.2. Загрузка нового документа

Метод `assign()` объекта `Location` заставляет окно загрузить и отобразить документ по указанному URL-адресу. Метод `replace()` выполняет похожую операцию, но перед открытием нового документа он удаляет текущий документ из списка посещавшихся страниц. Когда сценарию просто требуется загрузить новый документ, часто предпочтительнее использовать метод `replace()`, а не `assign()`. В противном случае кнопка Back (Назад) броузера вернет оригинальный документ и тот же самый сценарий снова загрузит новый документ. Метод `location.replace()` можно было бы использовать для загрузки версии веб-страницы со статической разметкой HTML, если сценарий обнаружит, что броузер пользователя не обладает функциональными возможностями, необходимыми для отображения полноценной версии:

```
// Если броузер не поддерживает объект XMLHttpRequest, выполнить
// переход к статической странице, которая не использует его.
if (!XMLHttpRequest) location.replace("staticpage.html");
```

Примечательно, что строка URL-адреса в этом примере, переданная методу `replace()`, представляет относительный адрес. Относительные URL-адреса интерпретируются относительно страницы, в которой они появляются, точно так же, как если бы они использовались в гиперссылке.

Кроме методов `assign()` и `replace()` объект `Location` определяет также метод `reload()`, который заставляет броузер перезагрузить документ.

Однако более традиционный способ заставить броузер перейти к новой странице заключается в том, чтобы просто присвоить новый URL-адрес свойству `location`:

```
location = "http://www.oreilly.com"; // Перейти, чтобы купить несколько книг!
```

Свойству `location` можно также присваивать относительные URL-адреса. Они разрешаются относительно текущего URL:

```
location = "page2.html"; // Загрузить следующую страницу
```

Идентификатор фрагмента – это особый вид относительного URL-адреса, который заставляет броузер просто прокрутить страницу, чтобы отобразить новый раздел, а не загружать новый документ. Идентификатор `#top` имеет специальное назначение: если в документе отсутствует элемент с идентификатором «top», он вынудит броузер перейти в начало документа:

```
location = "#top"; // Перейти в начало документа
```

Свойства декомпозиции URL объекта `Location` доступны для записи, и их изменение влечет за собой изменение URL-адреса в свойстве `location` и вынуждает броузер загрузить новый документ (или, в случае изменения свойства `hash`, выполнить переход внутри текущего документа):

```
location.search = "?page=" + (pagenum+1); // загрузить следующую страницу
```

14.3. История посещений

Свойство `history` объекта `Window` ссылается на объект `History` данного окна. Объект `History` хранит историю просмотра страниц в окне в виде списка документов и сведений о них. Свойство `length` объекта `History` позволяет узнать количество элементов в списке, но по причинам, связанным с безопасностью, сценарии не имеют возможности получить хранящиеся в нем URL-адреса. (Иначе любой сценарий смог бы исследовать историю посещения веб-сайтов.)

Методы `back()` и `forward()` действуют подобно кнопкам **Back (Назад)** и **Forward (Вперед)** броузера: они заставляют броузер перемещаться на один шаг назад и вперед по истории просмотра данного окна. Третий метод, `go()`, принимает целочисленный аргумент и пропускает заданное число страниц, двигаясь вперед (если аргумент положительный) или назад (если аргумент отрицательный) в списке истории.

```
history.go(-2); // Переход назад на 2 элемента, как если бы пользователь
                // дважды щелкнул на кнопке Back (Назад)
```

Если окно содержит дочерние окна (такие как элементы `<iframe>` – подробности смотрите в разделе 14.8.2), истории посещений в дочерних окнах хронологически чередуются с историей посещений в главном окне. То есть вызов `history.back()` (например) в главном окне может вызвать переход назад, к ранее отображавшемуся документу, в одном из дочерних окон, оставив главное окно в текущем состоянии.

Современные веб-приложения способны динамически изменять содержимое страницы без загрузки нового документа (примеры приводятся в главах 15 и 18). Приложениям, действующим подобным образом, может потребоваться предоставить пользователям возможность использовать кнопки Back и Forward для перехода между этими динамически созданными состояниями приложения. Стандарт HTML5 определяет два способа реализации, и оба они описываются в разделе 22.2.

Управление историей посещений до появления стандарта HTML5 представляло собой довольно сложную задачу. Приложение, управляющее собственной историей, должно было создавать новые записи в списке истории окна, связывать эти записи с информацией о состоянии, определять момент щелчка на кнопке Back, чтобы перейти к другой записи в списке, получать информацию, связанную с этой записью и воссоздавать предыдущее состояние приложения в соответствии с этой информацией. Один из приемов реализации такого поведения опирается на использование скрытого элемента `<iframe>`, в котором сохраняется информация о состоянии и создаются записи в списке истории просмотра. Чтобы создать новую запись, в этот скрытый фрейм динамически записывается новый документ, с помощью методов `open()` и `write()` объекта `Document` (раздел 15.10.2). Документ должен включать всю информацию, необходимую для воссоздания соответствующего состояния приложения. Когда пользователь щелкнет на кнопке Back, содержимое скрытого фрейма изменится. До появления стандарта HTML5 не предусматривалось никаких событий, которые извещали бы об этом изменении, поэтому, чтобы определить момент щелчка на кнопке Back, приходилось использовать функцию `setInterval()` (раздел 14.1) и с ее помощью 2–3 раза в секунду проверять наличие изменений в скрытом фрейме.

Однако на практике разработчики, когда требуется реализовать подобное управление историей просмотра, предпочитают использовать готовые решения. Многие фреймворки JavaScript включают такие решения. Например, для библиотеки jQuery существует расширение history. Существуют также автономные библиотеки управления историей. Например, одной из наиболее популярных является библиотека RSH (Really Simple History – действительно простое управление историей). Найти ее можно по адресу *http://code.google.com/p/reallysimplehistory/*. В разделе 22.2 описывается, как реализуется управление историей в HTML5.

14.4. Информация о броузере и об экране

Иногда сценариям бывает необходимо получить информацию о веб-броузере, в котором они выполняются, или об экране, на котором отображается броузер. В этом разделе описываются свойства `navigator` и `screen` объекта `Window`. Эти свойства ссылаются, соответственно, на объекты `Navigator` и `Screen`, содержащие информацию, которая дает возможность подстроить поведение сценария под существующее окружение.

14.4.1. Объект Navigator

Свойство `navigator` объекта `Window` ссылается на объект `Navigator`, содержащий общую информацию о номере версии и о производителе броузера. Объект `Navigator` назван «в честь» броузера Netscape Navigator, но он также поддерживается во всех других броузерах. (Кроме того, IE поддерживает свойство `clientInformation` как нейтральный синоним для `navigator`. К сожалению, другие броузеры свойство с таким именем не поддерживают.)

В прошлом объект `Navigator` обычно использовался сценариями для определения типа броузера – Internet Explorer или Netscape. Однако такой подход к определению типа броузера сопряжен с определенными проблемами, т. к. требует постоянного обновления с появлением новых броузеров или новых версий существующих броузеров. Ныне более предпочтительным считается метод на основе проверки функциональных возможностей (раздел 13.4.3). Вместо того чтобы делать какие-либо предположения о броузерах и их возможностях, гораздо проще прямо проверить наличие требуемой функциональной возможности (например, метода или свойства).

Однако иногда определение типа броузера может представлять определенную ценность. Один из таких случаев – возможность обойти ошибку, свойственную определенному типу броузера определенной версии. Объект `Navigator` имеет четыре свойства, предоставляющих информацию о версии работающего броузера, и вы можете использовать их для определения типа броузера:

`appName`

Название веб-броузера. В IE это строка «Microsoft Internet Explorer». В Firefox значением этого свойства является строка «Netscape». Для совместимости с существующими реализациями определения типа броузера значением этого свойства в других броузерах часто является строка «Netscape».

`appVersion`

Обычно значение этого свойства начинается с номера версии, за которым следует другая информация о версии броузера и его производителе. Обычно в начале строки указывается номер 4.0 или 5.0, свидетельствующий о совместимости с четвертым или пятым поколением броузеров. Формат строки в свойстве `appVersion` не определяется стандартом, поэтому невозможно организовать разбор этой строки способом, не зависящим от типа броузера.

`userAgent`

Строка, которую броузер посылает в http-заголовке `USER-AGENT`. Это свойство обычно содержит ту же информацию, что содержится в свойстве `appVersion`, а также может включать дополнительные сведения. Как и в случае со свойством `appVersion`, формат представления этой информации не стандартизован. Поскольку это свойство содержит больше информации, именно оно обычно используется для определения типа броузера.

`platform`

Строка, идентифицирующая операционную систему (и, возможно, аппаратную платформу), в которой работает броузер.

Сложность свойств объекта `Navigator` делает невозможной универсальную реализацию определения типа броузера. На раннем этапе развития Всемирной паутины было написано немало программного кода, зависящего от типа броузера, проверяющего свойства, такие как `navigator.appName`. Создавая новые броузеры, производители обнаружили, что для корректного отображения содержимого существующих веб-сайтов они должны устанавливать значение «Netscape» в свойстве `appName`. По тем же причинам потерял свою значимость номер в начале значения свойства appVersion, и в настоящее время реализация определения типа броузера должна опираться на строку в свойстве navigator.userAgent, имеющую более сложный формат, чем ранее. Пример 14.3 демонстрирует, как с помощью регулярных выражений (взятых из библиотеки jQuery) можно получить из свойства navigator.userAgent название броузера и номер версии.

Пример 14.3. Определение типа броузера с помощью свойства navigator.userAgent

```
// Определяет свойства browser.name и browser.version, позволяющие выяснить
// тип клиента. За основу взят программный код из библиотеки jQuery 1.4.1.
// Оба свойства, name и version, возвращают строки, и в обоих случаях
// значения могут отличаться от фактических названий броузеров и версий.
// Определяются следующие названия броузеров:
//
// "webkit": Safari или Chrome; version содержит номер сборки WebKit
// "opera": Opera; version содержит фактический номер версии броузера
// "mozilla": Firefox или другие броузеры, основанные на механизме gecko;
//            version содержит номер версии Gecko
// "msie": IE; version содержит фактический номер версии броузера
//
// Например, для Firefox 3.6 возвращаются: {name: "mozilla", version: "1.9.2"}
var browser = (function() {
    var s = navigator.userAgent.toLowerCase();
    var match = /(webkit)[ \/]([\w.]+)/.exec(s) ||
    /(opera)(?:.*version)?[ \/]([\w.]+)/.exec(s) ||
    /(msie) ([\w.]+)/.exec(s) ||
    !/compatible/.test(s) && /(mozilla)(?:.*? rv:([\w.]+))?/.exec(s) ||
     [];
    return { name: match[1] || "", version: match[2] || "0" };
}());
```

В дополнение к свойствам с информацией о версии и производителе броузера, объект Navigator имеет еще несколько свойств и методов. В число стандартных и часто реализуемых нестандартных свойств входят:

onLine

Свойство navigator.onLine (если существует) определяет, подключен ли броузер к сети. Приложениям может потребоваться сохранять информацию о состоянии локально (с использованием приемов, описываемых в главе 20), если броузер не подключен к сети.

geolocation

Объект Geolocation, определяющий API для выяснения географического положения пользователя. Подробнее об этом рассказывается в разделе 22.1.

javaEnabled()

Нестандартный метод, который должен возвращать true, если броузер способен выполнять Java-апплеты.

cookiesEnabled()

Нестандартный метод, который должен возвращать true, если броузер способен сохранять cookies. Если броузер настроен на сохранение cookies только для определенных сайтов, этот метод может возвращать некорректное значение.

14.4.2. Объект Screen

Свойство `screen` объекта `Window` ссылается на объект `Screen`, предоставляющий информацию о размере экрана на стороне пользователя и доступном количестве цветов. Свойства `width` и `height` возвращают размер экрана в пикселах. Свойства `availWidth` и `availHeight` возвращают фактически доступный размер экрана; из них исключается пространство, требуемое для таких графических элементов, как панель задач. Свойство `colorDepth` возвращает количество битов на пиксел, определяющих цвет. Типичными значениями являются 16, 24 и 32.

Свойство `window.screen` и объект `Screen`, на который оно ссылается, являются нестандартными, но они реализованы практически во всех броузерах. Объект `Screen` можно использовать, чтобы определить, не выполняется ли веб-приложение на устройстве с маленьким экраном, таком как нетбук. При ограниченном пространстве экрана, например, можно было бы использовать шрифты меньшего размера и маленькие изображения.

14.5. Диалоги

Объект `Window` обладает тремя методами для отображения простейших диалогов. Метод `alert()` выводит сообщение и ожидает, пока пользователь закроет диалоговое окно. Метод `confirm()` предлагает пользователю щелкнуть на кнопке OK или Cancel (Отмена) и возвращает логическое значение. Метод `prompt()` выводит сообщение, ждет ввода строки пользователем и возвращает эту строку. Ниже демонстрируется пример использования всех трех методов:

```
do {
    var name = prompt("Введите ваше имя");               // Вернет строку
    var correct = confirm("Вы ввели '" + name + "'.\n" + // Вернет логич. знач.
                          "Щелкните OK, чтобы продолжить, " +
                          "или Отмена, чтобы повторить ввод.");
} while(!correct)
alert("Привет, " + name); // Выведет простое сообщение
```

Методы `alert()`, `confirm()` и `prompt()` чрезвычайно просты в использовании, но правила хорошего дизайна требуют, чтобы они применялись как можно реже. Диалоги, подобные этим, нечасто используются в Веб, и большинство пользователей сочтет диалоговые окна, выводимые этими методами, выпадающими из обычной практики. Единственный вариант, когда имеет смысл обращаться к этим методам, – это отладка. JavaScript-программисты часто вставляют вызов метода `alert()` в программный код, пытаясь диагностировать возникшие проблемы.

Обратите внимание, что текст, отображаемый методами `alert()`, `confirm()` и `prompt()` в диалогах, – это обычный неформатированный текст. Его можно форматировать только пробелами, переводами строк и различными знаками пунктуации.

Методы `confirm()` и `prompt()` являются *блокирующими*, т. е. они не возвращают управление, пока пользователь не закроет отображаемые ими диалоговые окна.[1] Это значит, что, когда выводится одно из этих окон, программный код прекращает выполнение, и текущий загружаемый документ, если таковой существует, прекращает загружаться до тех пор, пока пользователь не отреагирует на запрос. В большинстве броузеров метод `alert()` также является блокирующим и ожидает от пользователя закрытия диалогового окна, но это не является обязательным требованием. Полное описание этих методов приводится в справочных статьях `Window.alert`, `Window.confirm` и `Window.prompt` в четвертой части книги.

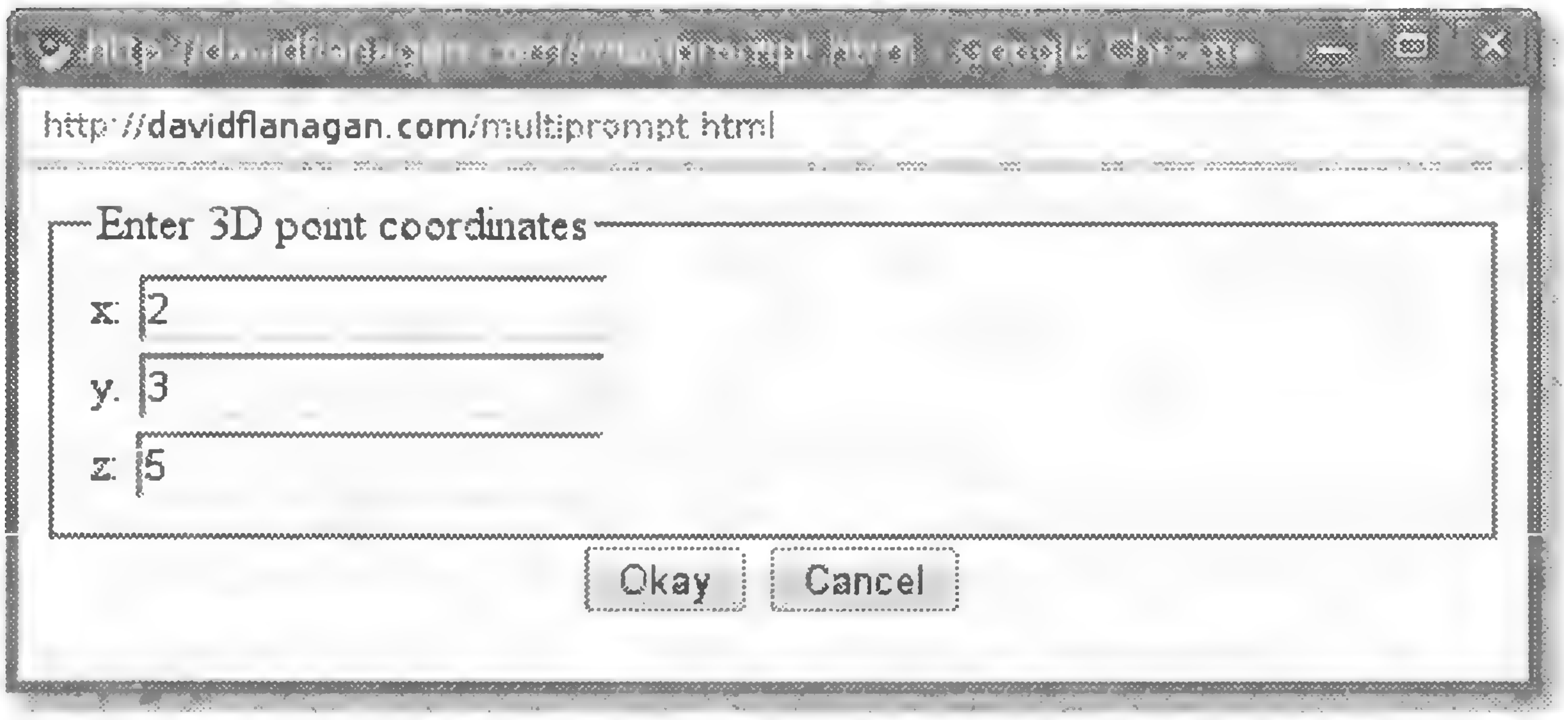
---

[1] Обычно такие окна называют модальными. – *Прим. науч. ред.*

В дополнение к методам `alert()`, `confirm()` и `prompt()` в объекте `Window` имеется более сложный метод, `showModalDialog()`, отображающий модальный диалог, содержащий разметку HTML, и позволяющий передавать аргументы и получать возвращаемое значение. Метод `showModalDialog()` выводит модальный диалог в отдельном окне броузера. Первым аргументом методу передается URL, определяющий HTML-содержимое диалога. Во втором аргументе может передаваться произвольное значение (допускается передавать массивы и объекты), которое будет доступно сценарию в диалоге, как значение свойства `window.dialogArguments`. Третий аргумент – нестандартный список пар имя/значение, разделенных точками с запятой, который, если поддерживается, может использоваться для настройки размеров и других атрибутов диалогового окна. Для определения размеров окна диалога можно использовать параметры «dialogwidth» и «dialogheight», а чтобы позволить пользователю изменять размеры окна, можно определить параметр «resizable=yes».

Окно, отображаемое эти методом, является модальным, и метод `showModalDialog()` не возвращает управление, пока окно не будет закрыто. После закрытия окна значение свойства `window.returnValue` становится возвращаемым значением метода. Обычно разметка HTML диалога должна включать кнопку `OK`, которая записывает желаемое значение в свойство `returnValue` и вызывает `window.close()` (раздел 14.8.1.1).

В примере 14.4 приводится разметка HTML для использования с методом `showModalDialog()`. Комментарий в начале примера включает пример вызова `showModalDialog()`, а на рис. 14.1 показан диалог, созданный вызовом из примера. Обратите внимание, что большая часть текста, отображаемого в диалоге, передается методу `showModalDialog()` во втором аргументе, а не является жестко определенной частью разметки HTML.


http://davidflanagan.com/multiprompt.html

Enter 3D point coordinates

x: 2

y: 3

z: 5

Рис. 14.1. Диалог, отображаемый методом showModalDialog()

Пример 14.4. HTML-файл для использования с функцией showModalDialog()

```
<!--
  Это не самостоятельный HTML-файл. Он должен вызываться методом showModalDialog()
  и ожидает получить в свойстве window.dialogArguments массив строк.
  Первый элемент массива - строка, отображаемая в верхней части диалога.
  Все остальные элементы - метки для однострочных текстовых полей ввода.
  Возвращает массив значений полей ввода после щелчка на кнопке Okay.
  Этот файл используется следующим образом:

  var p = showModalDialog("multiprompt.html",
                          ["Enter 3D point coordinates", "x", "y", "z"],
                           "dialogwidth:400; dialogheight:300; resizable:yes");
-->
<form>
<fieldset id="fields"></fieldset> <!-- Тело, заполняемое сценарием ниже -->
<div style="text-align:center">  <!-- Кнопки закрытия диалога -->
<button onclick="okay()">Okay</button>     <!-- Устанавливает возвращаемое -->
                                           <!-- значение и закрывает диалог -->
<button onclick="cancel()">Cancel</button> <!-- Закрывает диалог, -->
                                           <!-- не возвращая ничего -->
</div>
<script>
// Создает разметку HTML тела диалога и отображает ее в элементе fieldset
var args = dialogArguments;
var text = "<legend>" + args[0] + "</legend>";
for(var i = 1; i < args.length; i++)
    text += "<label>" + args[i] + ": <input id='f" + i + "'></label><br>";
document.getElementById("fields").innerHTML = text;

// Закрывает диалог без установки возвращаемого значения
function cancel() { window.close(); }

// Читает значения полей ввода и устанавливает возвращаемое значение,
// затем закрывает диалог
function okay() {
    window.returnValue = [];             // Возвращаемый массив
    for(var i = 1; i < args.length; i++) // Значения элементов из полей ввода
        window.returnValue[i-1] = document.getElementById("f" + i).value;
    window.close(); // Закрыть диалог. Это заставит showModalDialog() вернуть управление.
}
</script>
</form>
```

14.6. Обработка ошибок

Свойство onerror объекта Window – это обработчик событий, который вызывается во всех случаях, когда необработанное исключение достигло вершины стека вызовов и когда броузер готов отобразить сообщение об ошибке в консоли JavaScript. Если присвоить этому свойству функцию, функция будет вызываться всякий раз, когда в окне будет возникать ошибка выполнения программного кода JavaScript: присваиваемая функция станет обработчиком ошибок для окна.

Исторически сложилось так, что обработчику события onerror объекта Window передается три строковых аргумента, а не единственный объект события, как

в других обработчиках. (Другие объекты в клиентском JavaScript также имеют обработчики `onerror`, обрабатывающие различные ошибочные ситуации, но все они являются обычными обработчиками событий, которым передается единственный объект события.) Первый аргумент обработчика `window.onerror` – это сообщение, описывающее произошедшую ошибку. Второй аргумент – это строка, содержащая URL-адрес документа с JavaScript-кодом, приведшим к ошибке. Третий аргумент – это номер строки в документе, где произошла ошибка.

Помимо этих трех аргументов важную роль играет значение, возвращаемое обработчиком `onerror`. Если обработчик `onerror` возвращает `true`, это говорит броузеру о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, броузер не должен выводить собственное сообщение об ошибке. К сожалению, по историческим причинам в Firefox обработчик ошибок должен возвращать `true`, чтобы сообщить о том, что ошибка обработана.

Обработчик `onerror` является пережитком первых лет развития JavaScript, когда в базовом языке отсутствовала инструкция `try/catch` обработки исключений. В современном программном коде этот обработчик используется редко. Однако на время разработки вы можете определить свой обработчик ошибок, как показано ниже, который будет уведомлять вас о всех происходящих ошибках:

```
// Вывести сообщение об ошибке в виде диалога, но не более 3 раз
window.onerror = function(msg, url, line) {
    if (onerror.num++ < onerror.max) {
        alert("ОШИБКА: " + msg + "\n" + url + ":" + line);
        return true;
    }
}
onerror.max = 3;
onerror.num = 0;
```

14.7. Элементы документа как свойства окна

Если для именования элемента в HTML-документе используется атрибут `id` и если объект `Window` еще не имеет свойства, имя которого совпадает со значением этого атрибута, объект `Window` получает неперечислимое свойство с именем, соответствующим значению атрибута `id`, значением которого становится объект `HTMLElement`, представляющий этот элемент документа.

Как вы уже знаете, объект `Window` играет роль глобального объекта, находящегося на вершине цепочки областей видимости в клиентском JavaScript. Таким образом, вышесказанное означает, что атрибуты `id` в HTML-документах становятся глобальными переменными, доступными сценариям. Если, например, документ включает элемент `<button id="okay"/>`, на него можно сослаться с помощью глобальной переменной `okay`.

Однако важно отметить, что этого не происходит, если объект `Window` уже имеет свойство с таким именем. Элементы с атрибутами `id`, имеющими значение «history», «location» или «navigator», например, не будут доступны через глобальные переменные, потому что эти имена уже используются. Аналогично, если HTML-документ включает элемент с атрибутом `id`, имеющим значение «x» и в сценарии объявляется и используется глобальная переменная `x`, явно объявленная переменная скроет неявную переменную, ссылающуюся на элемент. Если переменная объявляется в сценарии, который в документе находится выше именованного элемента, наличие переменной будет препятствовать появлению нового свойства окна. А если переменная объявляется в сценарии, который находится ниже именованного элемента, первая же инструкция присваивания значения этой переменной затрет значение неявно созданного свойства.

В разделе 15.2 вы узнаете, что с помощью метода `document.getElementById()` можно отыскивать элементы документа по значениям их HTML-атрибутов `id`. Взгляните на следующий пример:

```
var ui = ["input","prompt","heading"];     // Массив идентификаторов элементов
ui.forEach(function(id) {                  // Отыскать элемент для каждого id
    ui[id] = document.getElementById(id);  // и сохранить его в свойстве
});
```

После выполнения этого фрагмента свойства `ui.input`, `ui.prompt` и `ui.heading` будут ссылаться на элементы документа. Вместо `ui.input` и `ui.heading` сценарий мог бы использовать глобальные переменные `input` и `heading`. Но, как вы помните из 14.5, объект `Window` имеет метод с именем `prompt()`, поэтому сценарий не сможет использовать глобальную переменную `prompt` вместо свойства `ui.prompt`.

Неявное использование идентификаторов элементов в качестве глобальных переменных – это пережиток истории развития веб-броузеров. Эта особенность необходима для сохранения обратной совместимости с существующими веб-страницами, но использовать ее сейчас не рекомендуется – в любой момент производители броузеров могут определить новое свойство в объекте `Window`, что нарушит работу любого программного кода, использующего неявно определяемое свойство с этим именем. Поиск элементов лучше выполнять явно, с помощью метода `document.getElementById()`. А его использование будет менее трудоемким, если дать ему более короткое имя:

```
var $ = function(id) { return document.getElementById(id); };
ui.prompt = $("prompt");
```

Многие клиентские библиотеки определяют функцию `$`, которая отыскивает элементы по их идентификаторам, как в примере выше. (В главе 19 мы узнаем, что функция `$` в библиотеке jQuery является универсальным методом выбора элементов, который может возвращать один или более элементов, опираясь на их идентификаторы, имена тегов, атрибут `class` или исходя из других критериев.)

Любой HTML-элемент с атрибутом `id` становится значением глобальной переменной, если значение атрибута `id` еще не используется объектом `Window`. Следующие HTML-элементы ведут себя подобным образом, если в них определен атрибут `name`:

```
<a> <applet> <area> <embed> <form> <frame> <frameset> <iframe> <img> <object>
```

Атрибут `id` элемента необходим для придания ему уникальности внутри документа: два элемента не могут иметь одинаковые значения атрибута `id`. Однако это не относится к атрибуту `name`. Если сразу несколько элементов, из перечисленных выше, будут иметь одно и то же значение атрибута `name` (или, если один элемент имеет атрибут `name`, а другой – атрибут `id` с тем же значением), неявно созданная глобальная переменная с этим именем будет ссылаться на объект, подобный массиву, хранящий все элементы с этим именем.

Элементы `<iframe>` с атрибутом `name` или `id` обрабатываются иначе. Переменная, неявно созданная для таких элементов, будет ссылаться не на объект `Element`, представляющий сам элемент, а на объект `Window`, представляющий вложенный фрейм, созданный элементом `<iframe>`. К этой теме мы еще вернемся в разделе 14.8.2.

14.8. Работа с несколькими окнами и фреймами

Единственное окно веб-броузера может содержать несколько вкладок. Каждая вкладка является независимым *контекстом просмотра*. Каждая имеет собственный объект `Window`, и каждая изолирована от всех остальных. Сценарий, выполняющийся в одной вкладке, обычно даже не имеет возможности узнать о существовании других вкладок, и тем более не имеет возможности взаимодействовать с их объектами `Window` или манипулировать содержащимися в них документами. Если вы пользуетесь броузером, который не поддерживает вкладки, или поддержка вкладок отключена, вы можете открыть сразу несколько окон веб-броузера. Как и в случае с вкладками, каждое окно имеет свой собственный объект `Window`, и каждое окно обычно изолировано и не зависит от всех остальных окон.

Но окна не всегда изолированы друг от друга. Сценарий в одном окне или вкладке может открывать новые окна или вкладки, и в этом случае окна могут взаимодействовать друг с другом и с находящимися в них документами (с учетом ограничений, накладываемых политикой общего происхождения, описываемой в разделе 13.6.2). Подробнее тема открытия и закрытия окон рассматривается в разделе 14.8.1.

HTML-документы могут содержать вложенные документы, используя для этого элементы `<iframe>`. Элемент `<iframe>` создает вложенный контекст просмотра, представленный отдельным объектом `Window`. Устаревшие и не рекомендуемые к использованию элементы `<frameset>` и `<frame>` также создают вложенные контексты просмотра, и каждый элемент `<frame>` представлен собственным объектом `Window`. Различия между окнами, вкладками, плавающими фреймами (элемент `<iframe>`) и фреймами в клиентском JavaScript весьма несущественны: все они являются отдельными контекстами просмотра и в сценариях все они представлены объектами `Window`. Вложенные контексты просмотра не изолированы друг от друга, как обычно бывают изолированы независимые вкладки. Сценарий, выполняющийся в одном фрейме, всегда имеет доступ к вмещающим и вложенным фреймам, и только политика общего происхождения может не позволять сценарию просматривать документы в этих фреймах. Вложенные фреймы рассматриваются в разделе 14.8.2.

Поскольку в клиентском JavaScript объект `Window` является глобальным объектом, каждое окно или фрейм имеет отдельный контекст выполнения сценариев на языке JavaScript. Однако сценарий JavaScript, выполняющийся в одном окне, может, с учетом ограничений политики общего происхождения, использовать объекты, свойства и методы, объявленные в другом окне. Более подробно эта тема обсуждается в разделе 14.8.3. На тот случай, когда политика общего происхождения не позволяет сценариям в двух отдельных окнах взаимодействовать друг с другом напрямую, стандарт HTML5 предусматривает прикладной интерфейс передачи сообщений, основанный на механизме событий, который обеспечивает возможность косвенного взаимодействия. Подробнее эта возможность рассматривается в разделе 22.3.

14.8.1. Открытие и закрытие окон

Открыть новое окно веб-броузера (или вкладку, что обычно зависит от настроек броузера) можно с помощью метода `open()` объекта `Window`. Метод `Window.open()` загружает документ по указанному URL-адресу в новое или в существующее окно и возвращает объект `Window`, представляющий это окно. Он принимает четыре необязательных аргумента:

Первый аргумент `open()` – это URL-адрес документа, отображаемого в новом окне. Если этот аргумент отсутствует (либо является пустой строкой), будет открыт специальный URL пустой страницы `about:blank`.

Второй аргумент `open()` – это строка с именем окна. Если окно с указанным именем уже существует (и сценарию разрешено просматривать содержимое этого окна), используется это существующее окно. Иначе создается новое окно и ему присваивается указанное имя. Если этот аргумент опущен, будет использовано специальное имя «_blank», т. е. будет открыто новое неименованное окно.

Обратите внимание, что сценарии не могут просто так указывать имена окон и не могут получать контроль над окнами, используемыми другими веб-приложениями: они могут указывать имена только тех существующих окон, которыми им «разрешено управлять» (термин взят из спецификации HTML5). Проще говоря, сценарий может указать имя существующего окна, только если это окно содержит документ, происходящий из того же источника, или если это окно было открыто самим сценарием (или рекурсивно открытым окном, которое открыло это окно). Кроме того, если одно окно является фреймом, вложенным в другое окно, сценарии в любом из них получают возможность управлять другим окном. В этом случае можно использовать зарезервированные имена «_top» (для обозначения вмещающего окна верхнего уровня) и «_parent» (для обозначения ближайшего вмещающего окна).

Имена окон

Имя окна играет важную роль, потому что оно позволяет указать существующее окно в вызове метода `open()`, а также потому, что оно может использоваться как значение HTML-атрибута `target` элементов `<a>` и `<form>`, ссылающихся на документ (или результат обработки формы), который должен быть отображен в именованном окне. Атрибуту `target` этих элементов можно также присвоить значение «_blank», «_parent» или «_top», чтобы открыть документ в новом пустом окне, родительском окне или фрейме или в окне верхнего уровня.

Имя окна, если оно имеется, хранится в свойстве `name` объекта `Window`. Данное свойство доступно для записи, и сценарии могут изменять его по мере необходимости. Если методу `Window.open()` передать имя (отличное от «_blank»), окно, созданное вызовом этого метода, получит указанное имя, как начальное значение свойства `name`. Если элемент `<iframe>` имеет атрибут `name`, объект `Window`, представляющий этот фрейм, будет использовать значение атрибута `name` как начальное значение свойства `name`.

Третий необязательный аргумент `open()` – это список параметров, определяющих размер и видимые элементы графического пользовательского интерфейса нового окна. Если опустить этот аргумент, окно получает размер по умолчанию и полный набор графических элементов: строку меню, строку состояния, панель инструментов и т. д. В броузерах, поддерживающих вкладки, это обычно приводит к созданию новой вкладки. Указав этот аргумент, можно явно определить размер окна и набор имеющихся в нем элементов управления. (Если явно указать размер, в большинстве случаев это приведет к созданию нового окна, а не вкладки.) Например, маленькое окно с изменяемым размером, имеющее строку состояния, но не содержащее меню, панели инструментов и адресную строку, можно открыть посредством следующим образом:

```
var w = window.open("smallwin.html", "smallwin",
                    "width=400,height=350,status=yes,resizable=yes");
```

Этот третий аргумент является нестандартным, и спецификация HTML5 требует, чтобы броузеры игнорировали его. Подробнее о том, какие параметры можно указывать в этом аргументе, рассказывается в описании метода `Window.open()` в четвертой части книги. Обратите внимание, что, когда указывается третий аргумент, любые не заданные явно элементы управления отсутствуют. По ряду причин, связанных с проблемами безопасности, броузеры накладывают ограничения на характеристики, которые можно передать методу. Так, например, невозможно открыть слишком маленькое окно или открыть его за пределами видимой области экрана; кроме того, некоторые броузеры не допускают возможности создания окон без строки состояния.

Указывать четвертый аргумент `open()` имеет смысл, только если второй аргумент определяет имя существующего окна. Этот аргумент – логическое значение, определяющее, должен ли URL-адрес, указанный в первом аргументе, заменить текущую запись в истории просмотра окна (`true`) или требуется создать новую запись (`false`). Если этот аргумент опущен, используется значение по умолчанию `false`.

Значение, возвращаемое методом `open()`, является объектом `Window`, представляющим вновь созданное окно. Этот объект позволяет сослаться в JavaScript-коде на новое окно так же, как исходный объект `Window` ссылается на окно, в котором выполняется сценарий:

```
var w = window.open();                           // Открыть новое пустое окно
w.alert("Будет открыт сайт http://example.com"); // Вызвать его метод alert()
w.location = "http://example.com";               // Установить св-во location
```

В окнах, созданных методом `window.open()`, свойство `opener` ссылается на объект `Window` сценария, открывшего его. В других случаях свойство `opener` получает значение `null`:

```
w.opener !== null;     // Верно для любого окна w, созданного методом open()
w.open().opener === w; // Верно для любого окна w
```

Метод `Window.open()` часто используется рекламодателями для создания «всплывающих окон» с рекламой, когда пользователь путешествует по Всемирной паутине. Такие всплывающие окна могут раздражать пользователя, поэтому большинство веб-броузеров реализуют механизм блокирования всплывающих окон. Обычно вызов метода `open()` преуспевает, только если он производится в ответ на действия пользователя, такие как щелчок мышью на кнопке или на ссылке. Попытка открыть всплывающее окно, которая производится, когда броузер просто загружает (или выгружает) страницу, обычно оканчивается неудачей. Попытка протестировать представленные выше строки программного кода в консоли JavaScript вашего броузера также может окончиться неудачей по тем же причинам.

14.8.1.1. Закрытие окон

Новое окно открывается при помощи метода `open()` и закрывается при помощи метода `close()`. Если объект `Window` был создан сценарием, то этот же сценарий сможет закрыть его следующей инструкцией:

```
w.close();
```

JavaScript-код, выполняющийся внутри данного окна, может закрыть его так:

```
window.close();
```

Обратите внимание на явное использование идентификатора `window` для устранения неоднозначности между методом `close()` объекта `Window` и методом `close()` объекта `Document` – это важно, если метод `close()` вызывается внутри обработчика события.

Большинство броузеров разрешают программисту автоматически закрывать только те окна, которые были созданы его собственным JavaScript-кодом. Если сценарий попытается закрыть любое другое окно, появится диалоговое окно с запросом к пользователю подтвердить (или отменить) закрытие окна. Вызов метода `close()` объекта `Window`, представляющего фрейм, а не окно верхнего уровня или вкладку, не выполняет никаких действий: нельзя закрыть фрейм (можно только удалить элемент `<iframe>` из содержащего его документа).

Объект `Window` продолжает существовать и после закрытия представляемого им окна. Однако не следует использовать какие-либо его свойства или методы, исключая проверку свойства `closed`. Если окно было закрыто, это свойство будет иметь значение `true`, свойство `document` – значение `null`, а методы окна обычно не выполняются.

14.8.2. Отношения между фреймами

Мы уже видели, что метод `open()` объекта `Window` возвращает новый объект `Window`, свойство `opener` которого ссылается на первоначальное окно. Таким образом, два окна могут ссылаться друг на друга, и каждое из них может читать свойства и вызывать методы другого. То же самое возможно для фреймов. Сценарий, выполняющийся в окне или фрейме, может ссылаться на объемлющее или вложенное окно или фрейм при помощи свойств, описываемых ниже.

Как вы уже знаете, сценарий в любом окне или фрейме может сослаться на собственное окно или фрейм с помощью свойства `window` или `self`. Фрейм может сослаться на объект `Window` вмещающего окна или фрейма с помощью свойства `parent`:

```
parent.history.back();
```

Объект `Window`, представляющий окно верхнего уровня или вкладку, не имеет вмещающего окна, поэтому его свойство `parent` просто ссылается на само окно:

```
parent == self; // Для любых окон верхнего уровня
```

Если фрейм находится внутри другого фрейма, содержащегося в окне верхнего уровня, то он может сослаться на окно верхнего уровня так: `parent.parent`. Однако в качестве универсального сокращения имеется свойство `top`: независимо от глубины вложенности фрейма его свойство `top` ссылается на содержащее его окно самого верхнего уровня. Если объект `Window` представляет окно верхнего уровня, свойство `top` просто ссылается на само окно. Для фреймов, непосредственно принадлежащих окну верхнего уровня, значение свойства `top` совпадает со значением свойства `parent`.

Свойства `parent` и `top` позволяют сценариям ссылаться на родительские окна или фреймы. Существует несколько способов сослаться на дочерние окна или фреймы. Фреймы создаются с помощью элементов `<iframe>`. Получить ссылку на объект `Element`, представляющий элемент `<iframe>`, можно точно так же, как на объект, представляющий любой другой элемент. Допустим, что документ содержит тег `<iframe id="f1">`. Тогда получить ссылку на объект `Element`, представляющий этот элемент `iframe` можно следующим образом:

```
var iframeElement = document.getElementById("f1");
```

Элементы `<iframe>` имеют свойство `contentWindow`, которое ссылается на объект `Window` фрейма, поэтому ссылку на объект `Window` этого фрейма можно получить так:

```
var childFrame = document.getElementById("f1").contentWindow;
```

Имеется возможность пойти обратным путем – от объекта `Window`, представляющего фрейм, к объекту `Element` элемента `<iframe>`, содержащего фрейм, – с помощью свойства `frameElement` объекта `Window`. Объекты `Window`, представляющие окна верхнего уровня, а не фреймы, имеют значение `null` в свойстве `frameElement`:

```
var elt = document.getElementById("f1");
var win = elt.contentWindow;
win.frameElement === elt      // Всегда верно для фреймов
window.frameElement === null  // Для окон верхнего уровня
```

Однако, чтобы получить ссылки на дочерние фреймы, обычно не требуется использовать метод `getElementById()` и свойство `contentWindow`. Каждый объект `Window` имеет свойство `frames`, хранящее ссылки на дочерние фреймы, содержащиеся в окне или фрейме. Свойство `frames` ссылается на объект, подобный массиву, который может индексироваться числовыми индексами или именами фреймов. Получить ссылку на первый дочерний фрейм в окне можно с помощью выражения `frames[0]`. Сослаться на третий дочерний фрейм во втором дочернем фрейме можно с помощью выражения `frames[1].frames[2]`. Сценарий, выполняющийся во фрейме, может сослаться на соседний фрейм одного с ним уровня, как `parent.frames[1]`. Обратите внимание, что элементами массива `frames[]` являются объекты `Window`, а не элементы `<iframe>`.

Если в элементе `<iframe>` указать атрибут `name` или `id`, в качестве индекса этого фрейма можно будет использовать не только число, но и имя. Например, ссылку на фрейм с именем «**f1**» можно получить с помощью выражения `frames["f1"]` или `frames.f1`.

В разделе 14.7 говорилось, что имена или идентификаторы элементов `<iframe>` и других автоматически превращаются в свойства объекта `Window` и что элементы

`<iframe>` интерпретируются иначе, чем другие элементы: в случае с фреймами значениями этих автоматически создаваемых свойств становятся ссылки на объекты `Window`, а не на объекты `Element`. Это означает, что на фрейм с именем «f1» можно сослаться как на свойство `f1` вместо `frames.f1`. В действительности, стандарт HTML5 указывает, что свойство `frames`, подобно свойствам `window` и `self`, ссылается на сам объект `Window`, который действует как массив фреймов. Это означает, что на первый дочерний фрейм можно сослаться, как `window[0]`, а получить количество фреймов можно, обратившись к свойству `window.length` или просто `length`. Однако использование свойства `frames` вместо `window` в подобных случаях делает программный код более понятным. Обратите внимание, что не во всех текущих броузерах выполняется условие `frame==window`, но даже в броузерах, где это условие не выполняется, разрешается индексировать дочерние фреймы числами и именами, обращаясь к любому из этих двух объектов.

С помощью атрибута `name` или `id` элементу `<iframe>` можно присвоить имя, которое будет доступно для использования в JavaScript-коде. Однако если использовать атрибут `name`, указанное имя также будет использоваться в качестве значения свойства `name` объекта `Window`, представляющего фрейм. Имя, указанное таким способом, можно использовать в качестве значения атрибута `target` ссылки и передавать методу `window.open()` во втором аргументе.

14.8.3. JavaScript во взаимодействующих окнах

Для каждого окна или фрейма имеется свой собственный объект `Window`, определяющий независимый контекст выполнения JavaScript-кода. Но если сценарий в одном окне или фрейме может сослаться на другое окно или фрейм (и если политика общего происхождения не препятствует этому), сценарий в одном окне или фрейме может взаимодействовать со сценарием в другом окне или фрейме.

Представим себе веб-страницу с двумя элементами `<iframe>`, с именами «A» и «B», и предположим, что эти фреймы содержат документы, полученные с одного и того же сервера, и эти документы содержат взаимодействующие сценарии. Сценарий во фрейме A определяет переменную `i`:

```
var i = 3;
```

Это переменная представляет собой свойство глобального объекта, т. е. свойство объекта `Window`. Сценарий во фрейме A может явно сослаться на эту переменную как на свойство с помощью объекта `window`:

```
window.i
```

Благодаря тому что сценарий во фрейме B может ссылаться на объект `Window` во фрейме A, он также может ссылаться на свойства этого объекта окна:

```
parent.A.i = 4;    // Изменит значение переменной во фрейме A
```

Напомню, что ключевое слово `function`, определяющее функцию, объявляет переменную так же, как ключевое слово `var`. Если JavaScript-код во фрейме B объявляет функцию `f`, эта функция станет глобальной переменной во фрейме B, и сценарий во фрейме B сможет вызывать функцию `f` как `f()`. Однако сценарий во фрейме A должен ссылаться на `f` как на свойство объекта `Window` во фрейме B:

```
parent.B.f();      // Вызовет функцию, объявленную во фрейме B
```

Если сценарий во фрейме А часто вызывает эту функцию, ее можно присвоить переменной во фрейме А, чтобы было удобнее ссылаться на функцию:

```
var f = parent.B.f;
```

Теперь сценарий во фрейме А сможет вызывать функцию как `f()` точно так же, как сценарий во фрейме В.

Разделяя подобным образом функции между фреймами или окнами, очень важно помнить о правилах лексического контекста. Функции выполняются в том контексте, в котором они определены, а не в том, из которого они вызываются. Следовательно, если функция `f` ссылается на глобальные переменные, поиск этих переменных выполняется в свойствах фрейма В, даже когда функция вызывается из фрейма А.

Напомню, что конструкторы – это тоже функции, поэтому когда вы определяете класс объектов (см. главу 9) с функцией-конструктором и связанным с ним объектом-прототипом, этот класс будет определен только для одного окна. Предположим, что окно, содержащее фреймы А и В, включает класс `Set` из примера 9.6.

Сценарии в окне верхнего уровня смогут создавать новые объекты `Set`, как показано ниже:

```
var s = new Set();
```

Но сценарии в обоих фреймах должны явно ссылаться на конструктор `Set()`, как на свойство родительского окна:

```
var s = new parent.Set();
```

В качестве альтернативы сценарий в любом фрейме может определить собственные переменные для более удобного обращения к функции-конструктору:

```
var Set = top.Set();
var s = new Set();
```

В отличие от пользовательских классов, предопределенные классы, такие как `Set`, `Date` и `RegExp`, оказываются автоматически определенными во всех окнах. Однако следует заметить, что каждое окно имеет независимую копию конструктора и независимую копию объекта-прототипа. Например, каждое окно имеет собственную копию конструктора `String()` и объекта `String.prototype`. Поэтому, если вы создадите новый метод для работы с JavaScript-строками и сделаете его методом класса `String`, присвоив его объекту `String.prototype` в текущем окне, все строки в этом окне смогут использовать новый метод, однако этот новый метод будет недоступен строкам, определенным в других окнах.

Тот факт, что каждый объект `Window` имеет собственные объекты-прототипы, означает, что оператор `instanceof` не будет работать с объектами в разных окнах. Например, оператор `instanceof` будет возвращать `false` при сопоставлении строки из фрейма В с конструктором `String()` из фрейма А. В разделе 7.10 описываются похожие сложности с определением типов массивов в разных окнах.

Объект WindowProxy

Мы неоднократно отмечали, что в клиентском JavaScript объект `Window` является глобальным объектом. Однако если смотреть с технической точки зрения, это не так. Каждый раз, когда веб-броузер загружает содержимое в окно или фрейм, он должен создать новый контекст выполнения JavaScript, включая и новый глобальный объект. Но при наличии нескольких взаимодействующих окон или фреймов очень важно обеспечить сохранность ссылки на объект `Window`, представляющий фрейм или окно, даже если в этот фрейм или окно загружается новый документ.

Таким образом, в клиентском JavaScript имеется два объекта, играющих важную роль. Первый – это клиентский глобальный объект, который находится на вершине цепочки областей видимости и в котором определяются глобальные переменные и функции. Этот глобальный объект действительно замещается новым объектом, когда в окно или фрейм загружается новый документ. Второй объект, который мы называем объектом `Window`, в действительности не является глобальным объектом – это промежуточный объект. Всякий раз, когда сценарий читает или изменяет значение свойства объекта `Window`, этот объект запрашивает или изменяет свойство с тем же именем глобального объекта окна или фрейма. В спецификации HTML5 этот промежуточный объект называется объектом `WindowProxy`, но далее в этой книге мы продолжим использовать имя `Window`.

Этот промежуточный объект ведет себя как настоящий глобальный объект, за исключением того, что живет гораздо дольше. Если бы вы могли сравнить эти два объекта, вы едва ли смогли бы отличить их. Однако на самом деле нет никакой возможности сослаться на настоящий клиентский глобальный объект. Глобальный объект находится на вершине цепочки областей видимости, но свойства `window`, `self`, `top`, `parent` и `frames` ссылаются на промежуточные объекты. Метод `window.open()` возвращает промежуточный объект. Даже ключевое слово `this` в функциях верхнего уровня ссылается на промежуточный объект, а не на настоящий глобальный объект.[1]

---

[1] Этот последний пункт является небольшим отступлением от стандартов ES3 и ES5, но это необходимо для поддержки взаимодействующих контекстов выполнения в клиентском JavaScript.

15

Работа с документами

Клиентский JavaScript предназначен для того, чтобы превращать статические HTML-документы в интерактивные веб-приложения. Работа с содержимым веб-страниц – главное предназначение JavaScript. Данная глава является одной из наиболее важных в этой книге – здесь рассказывается о том, как это делается.

В главах 13 и 14 говорилось, что каждое окно, вкладка и фрейм веб-броузера представлено объектом `Window`. Каждый объект `Window` имеет свойство `document`, ссылающееся на объект `Document`. Этот объект `Document` и является темой обсуждения данной главы. Однако объект `Document` не является автономным объектом. Он является центральным объектом обширного API, известного как *объектная модель документа* (Document Object Model, DOM), который определяет порядок доступа к содержимому документа.

Эта глава начинается с описания базовой архитектуры DOM, а затем она расскажет:

- Как выбирать отдельные элементы документа.
- Как выполняется *обход* содержимого документа, представленного в виде дерева узлов, и как отыскивать в нем родительские, дочерние и братские элементы.
- Как читать и изменять значения атрибутов элементов документа.
- Как читать и изменять содержимое документа.
- Как изменять структуру документа, создавая, вставляя и удаляя узлы.
- Как работать с HTML-формами.

В заключительном разделе этой главы рассматриваются различные особенности документов, включая свойство `referrer`, метод `write()` и приемы получения текста, выделенного в документе.

15.1. Обзор модели DOM

Объектная модель документа (Document Object Model, DOM) – это фундаментальный прикладной программный интерфейс, обеспечивающий возможность работы с содержимым HTML- и XML-документов. Прикладной программный интерфейс (API) модели DOM не особенно сложен, но в нем существует множество архитектурных особенностей, которые вы должны знать

Прежде всего, следует понимать, что вложенные элементы HTML- или XML-документов представлены в виде дерева объектов DOM. Древовидное представление HTML-документа содержит узлы, представляющие элементы или теги, такие как `<body>` и `<p>`, и узлы, представляющие строки текста. HTML-документ также может содержать узлы, представляющие HTML-комментарии. Рассмотрим следующий простой HTML-документ:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
</html>
```

DOM-представление этого документа приводится на рис. 15.1.

Тем, кто еще не знаком с древовидными структурами в компьютерном программировании, полезно узнать, что терминология для их описания была заимствована у генеалогических деревьев. Узел, расположенный непосредственно над данным узлом, называется *родительским* по отношению к данному узлу. Узлы, расположенные на один уровень ниже другого узла, являются *дочерними* по отношению к данному узлу. Узлы, находящиеся на том же уровне и имеющие того же родителя, называются *братьями*. Узлы, расположенные на любое число уровней ниже другого узла, являются его *потомками*. Родительские, прародительские и любые другие узлы, расположенные выше данного узла, являются его *предками*.

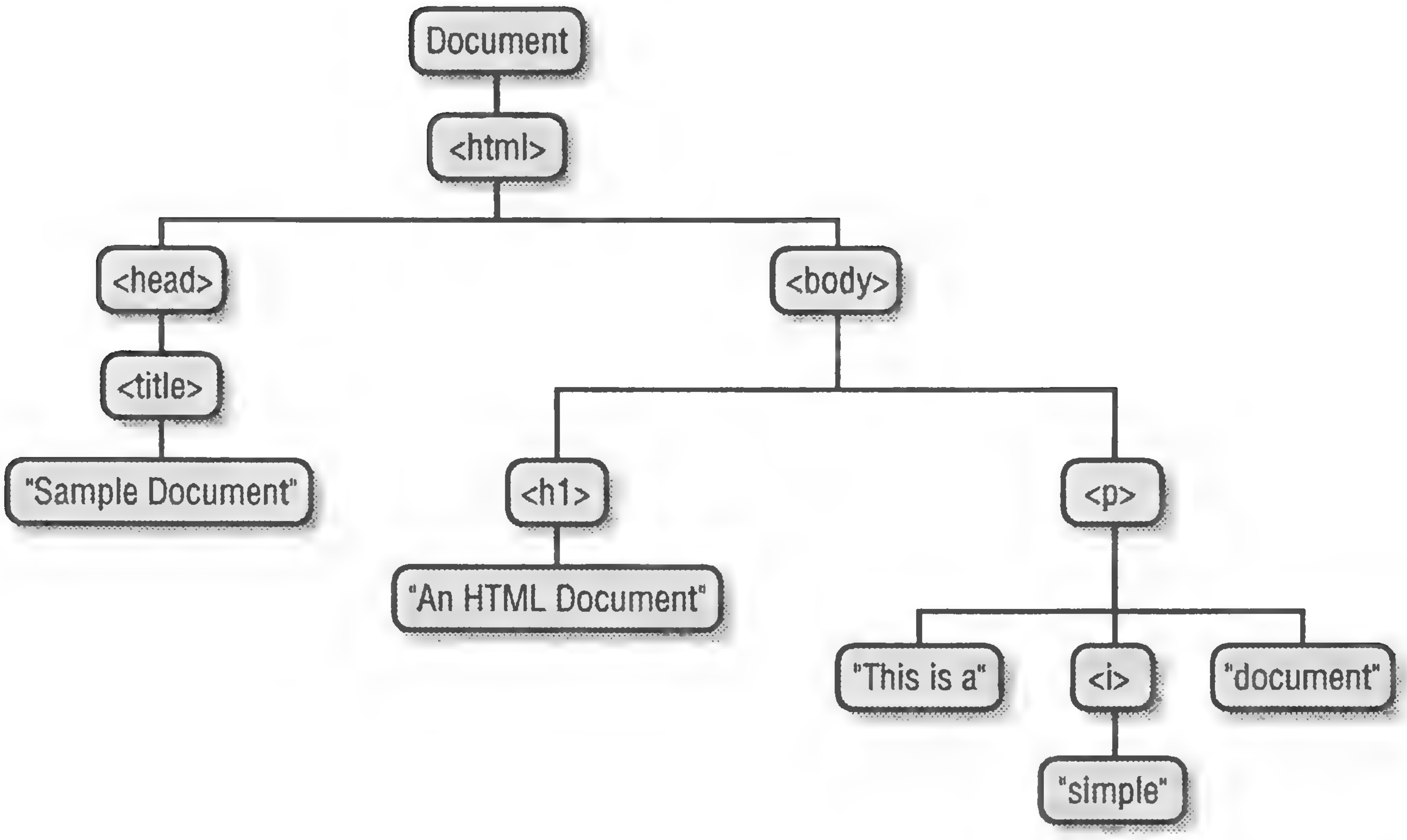


Рис. 15.1. Древовидное представление HTML-документа

Каждый прямоугольник на рис. 15.1 является узлом документа, который представлен объектом Node. О свойствах и методах объекта Node будет рассказываться в некоторых разделах, следующих ниже, кроме того, описания этих свойств вы найдете в справочной статье Node в четвертой части книги. Обратите внимание, что на рисунке изображено три различных типа узлов. Корнем дерева является узел Document, который представляет документ целиком. Узлы, представляющие HTML-элементы, являются узлами типа Element, а узлы, представляющие текст, – узлами типа Text. Document, Element и Text – это подклассы класса Node, и для них имеются отдельные справочные статьи в четвертой части книги. Document и Element являются двумя самыми важными классами в модели DOM, и большая часть главы посвящена знакомству с их свойствами и методами.

Тип Node и его подтипы образуют иерархию типов, изображенную на рис. 15.2. Обратите внимание на формальные отличия между обобщенными типами Document и Element, и типами HTMLDocument и HTMLElement. Тип Document представляет HTML- и XML-документ, а класс Element представляет элемент этого документа. Подклассы HTMLDocument и HTMLElement представляют конкретно HTML-документ и его элементы. В этой книге часто используются имена обобщенных классов Document и Element, даже когда подразумеваются HTML-документы. То же самое относится и к справочному разделу книги: свойства и методы типов HTMLDocument и HTMLElement описываются в справочных статьях Document и Element.

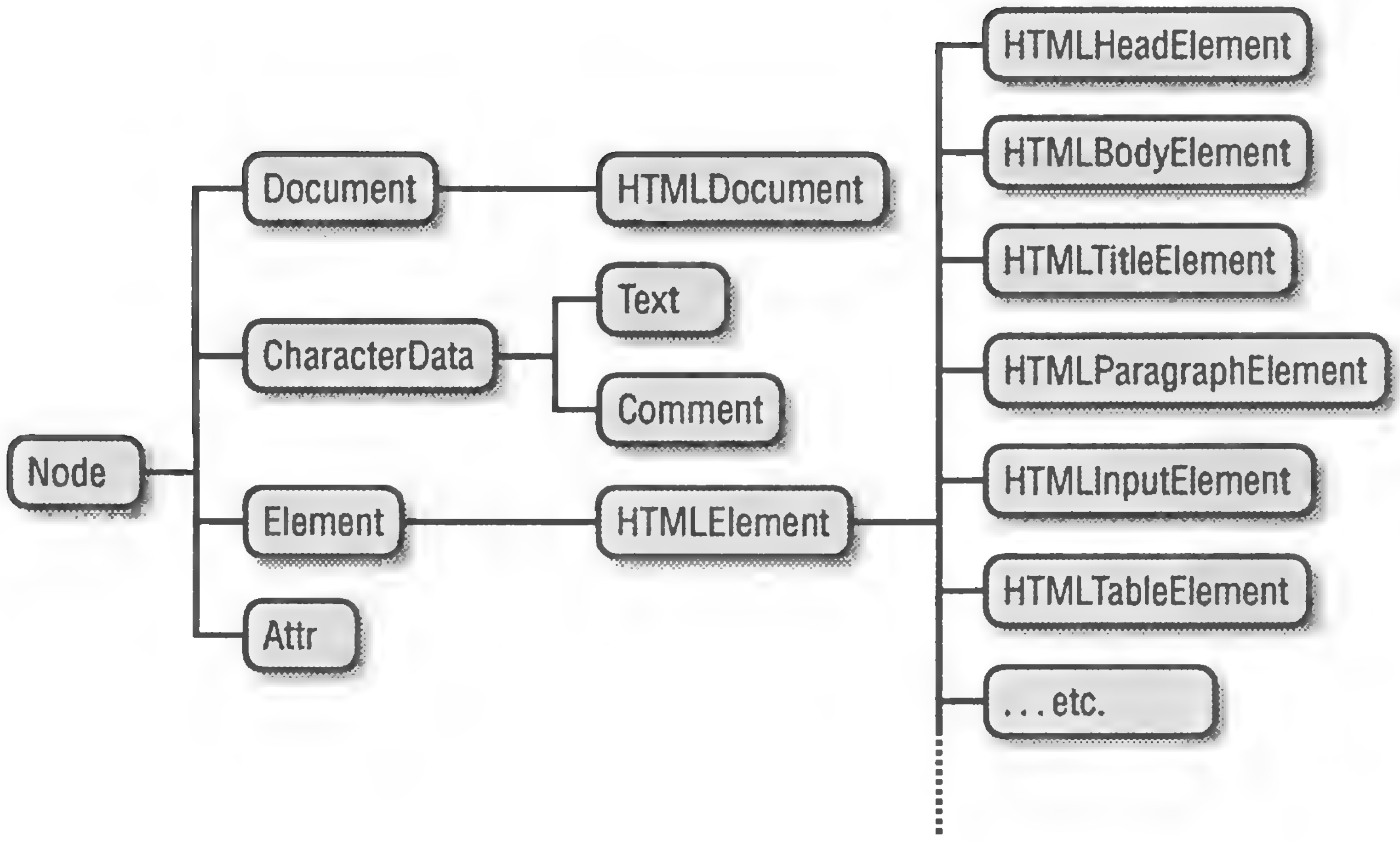


Рис. 15.2. Неполная иерархия классов узлов документов

На рис. 15.2 следует также отметить наличие большого количества подтипов класса HTMLElement, представляющих конкретные типы HTML-элементов. Каждый из них определяет JavaScript-свойства, отражающие HTML-атрибуты конкретного элемента или группы элементов (раздел 15.4.1). Некоторые из этих специфических классов определяют дополнительные свойства или методы, которые не являются отражением синтаксиса языка разметки HTML. Более подробно эти классы и их дополнительные особенности рассматриваются в справочном разделе книги.

Наконец, обратите внимание, что на рис. 15.2 изображены некоторые типы узлов, которые нигде до сих пор не упоминались. Узлы `Comment` представляют HTML- или XML-комментарии. Поскольку комментарии являются обычными текстовыми строками, эти узлы во многом подобны узлам `Text`, представляющим отображаемый текст документа. Тип `CharacterData`, обобщенный предок типов `Text` и `Comment`, определяет методы, общие для узлов этих двух типов. Узлы типа `Attr` представляют XML- или HTML-атрибуты, но они практически никогда не используются, потому что класс `Element` определяет методы, позволяющие интерпретировать атрибуты, как пары имя/значение, а не как узлы документа. Объект `DocumentFragment` (не изображен на рисунке) – это разновидность узлов, которая практически никогда не встречается в документах: он представляет последовательность узлов, не имеющих общего родителя. Объекты `DocumentFragment` удобно использовать при выполнении манипуляций над документами, и подробнее об этом типе рассказывается в разделе 15.6.4. Модель DOM также определяет несколько редко используемых типов, представляющих, например, объявления типа документа и инструкции обработки XML.

15.2. Выбор элементов документа

Работа большинства клиентских программ на языке JavaScript так или иначе связана с манипулированием элементами документа. В ходе выполнения эти программы могут использовать глобальную переменную `document`, ссылающуюся на объект `Document`. Однако, чтобы выполнить какие-либо манипуляции с элементами документа, программа должна каким-то образом получить, или *выбрать*, объекты `Element`, ссылающиеся на эти элементы документа. Модель DOM определяет несколько способов выборки элементов. Выбрать элемент или элементы документа можно:

- по значению атрибута `id`;
- по значению атрибута `name`;
- по имени тега;
- по имени класса или классов CSS; или
- по совпадению с определенным селектором CSS.

Все эти приемы выборки элементов описываются в следующих подразделах.

15.2.1. Выбор элементов по значению атрибута id

Все HTML-элементы имеют атрибуты `id`. Значение этого атрибута должно быть уникальным в пределах документа – никакие два элемента в одном и том же документе не должны иметь одинаковые значения атрибута `id`. Выбрать элемент по уникальному значению атрибута `id` можно с помощью метода `getElementById()` объекта `Document`. Этот метод уже использовался в примерах глав 13 и 14:

```
var section1 = document.getElementById("section1");
```

Это самый простой и самый распространенный способ выборки элементов. Если сценарию необходимо иметь возможность манипулировать каким-то определенным множеством элементов документа, присвойте значения атрибутам `id` этих элементов и используйте возможность их поиска по этим значениям. Если потре-

буется отыскать более одного элемента по значению атрибута `id`, можно воспользоваться удобной функцией `getElements()`, реализация которой приводится в примере 15.1.

Пример 15.1. Поиск нескольких элементов по значениям атрибута id

```
/**
 * Эта функция принимает произвольное количество строковых аргументов.
 * Каждый аргумент интерпретируется как значение атрибута id элемента,
 * и для каждого из них вызывается метод document.getElementById().
 * Возвращает объект, который отображает значения атрибута id
 * в соответствующие объекты Element. Если какое-то значение атрибута id
 * не будет найдено в документе, возбуждает исключение Error.
 */
function getElements(/* значения атрибутов id...*/) {
    var elements = {};                          // Создать пустое отображение
    for(var i = 0; i < arguments.length; i++) { // Для каждого аргумента
        var id = arguments[i];                  // Аргумент - id элемента
        var elt = document.getElementById(id);  // Отыскать элемент
        if (elt == null)                        // Если не найден,
            throw new Error("No element with id: " + id); // возбудить ошибку
        elements[id] = elt;                     // Отобразить id в элемент
    }
    return elements; // Вернуть отображение id в элементы
}
```

В версиях Internet Explorer ниже IE8 метод `getElementById()` выполняет поиск значений атрибутов `id` без учета регистра символов и, кроме того, возвращает элементы, в которых будет найдено совпадение со значением атрибута `name`.

15.2.2. Выбор элементов по значению атрибута name

HTML-атрибут `name` первоначально предназначался для присваивания имен элементам форм, и значение этого атрибута использовалось, когда выполнялась отправка данных формы на сервер. Подобно атрибуту `id`, атрибут `name` присваивает имя элементу. Однако, в отличие от `id`, значение атрибута `name` не обязано быть уникальным: одно и то же имя могут иметь сразу несколько элементов, что вполне обычно при использовании в формах радиокнопок и флажков. Кроме того, в отличие от `id`, атрибут `name` допускается указывать лишь в некоторых HTML-элементах, включая формы, элементы форм и элементы `<iframe>` и `<img>`.

Выбрать HTML-элементы, опираясь на значения их атрибутов `name`, можно с помощью метода `getElementsByName()` объекта `Document`:

```
var radiobuttons = document.getElementsByName("favorite_color");
```

Метод `getElementsByName()` определяется не классом `Document`, а классом `HTMLDocument`, поэтому он доступен только в HTML-документах и не доступен в XML-документах. Он возвращает объект `NodeList`, который ведет себя, как доступный только для чтения массив объектов `Element`. В IE метод `getElementsByName()` возвращает также элементы, значения атрибутов `id` которых совпадает с указанным значением. Чтобы обеспечить совместимость с разными версиями броузеров, необходимо внимательно подходить к выбору значений атрибутов и не использовать одни и те же строки в качестве значений атрибутов `name` и `id`.

Мы видели в разделе 14.7, что наличие атрибута `name` в некоторых HTML-элементах приводит к автоматическому созданию свойств с этими именами в объекте `Window`. То же относится и к объекту `Document`. Наличие атрибута `name` в элементе `<form>`, `<img>`, `<iframe>`, `<applet>`, `<embed>` или `<object>` (но только в элементе `<object>`, который не имеет вложенных объектов с альтернативным содержимым) приводит к созданию свойства в объекте `Document`, имя которого совпадает со значением атрибута (при этом предполагается, что объект документа еще не имеет свойства с этим именем).

Если существует только один элемент с указанным именем, значением автоматически созданного свойства документа станет сам элемент. Если таких элементов несколько, значением свойства будет объект `NodeList`, играющий роль массива элементов. Как было показано в разделе 14.7, для именованных элементов `<iframe>` создаются особые свойства документа: они ссылаются не на объекты `Element`, а на объекты `Window`, представляющие фреймы.

Это означает, что некоторые элементы могут быть выбраны по их именам простым обращением к свойствам объекта `Document`:

```
// Получить ссылку на объект Element для элемента <form name="shipping_address">
var form = document.shipping_address;
```

Причины, почему не следует использовать автоматически создаваемые свойства окна, которые описываются в разделе 14.7, в равной степени применимы и к автоматически создаваемым свойствам документа. Если вам потребуется отыскать именованные элементы, лучше всего это сделать явно, с помощью метода `getElementsByName()`.

15.2.3. Выбор элементов по типу

Метод `getElementsByTagName()` объекта `Document` позволяет выбрать все HTML- или XML-элементы указанного типа (или по имени тега). Например, получить подобный массиву объект, доступный только для чтения, содержащий объекты `Element` всех элементов `<span>` в документе, можно следующим образом:

```
var spans = document.getElementsByTagName("span");
```

Подобно методу `getElementsByName()`, `getElementsByTagName()` возвращает объект `NodeList`. (Подробнее класс `NodeList` описывается во врезке, в этом же разделе.) Элементы документа включаются в массив `NodeList` в том же порядке, в каком они следуют в документе, т. е. первый элемент `<p>` в документе можно выбрать так:

```
var firstpara = document.getElementsByTagName("p")[0];
```

Имена HTML-тегов не чувствительны к регистру символов, и когда `getElementsByTagName()` применяется к HTML-документу, он выполняет сравнение с именем тега без учета регистра символов. Переменная `spans`, созданная выше, например, будет включать также все элементы `<span>`, которые записаны как `<SPAN>`.

Можно получить `NodeList`, содержащий все элементы документа, если передать методу `getElementsByTagName()` шаблонный символ «*».

Кроме того, классом `Element` также определяет метод `getElementsByTagName()`. Он действует точно так же, как и версия метода в классе `Document`, но выбирает только элементы, являющиеся потомками для элемента, относительно которого вызывается метод. То есть отыскать все элементы `<span>` внутри первого элемента `<p>` можно следующим образом:

```
var firstpara = document.getElementsByTagName("p")[0];
var firstParaSpans = firstpara.getElementsByTagName("span");
```

По историческим причинам класс `HTMLDocument` определяет специальные свойства для доступа к узлам определенных типов. Свойства `images`, `forms` и `links`, например, ссылаются на объекты, которые ведут себя как массивы, доступные только для чтения, содержащие элементы `<img>`, `<form>` и `<a>` (но только те теги `<a>`, которые имеют атрибут `href`). Эти свойства ссылаются на объекты `HTMLCollection`, которые во многом похожи на объекты `NodeList`, но дополнительно могут индексироваться значениями атрибутов `id` и `name`. Ранее мы узнали, как можно получить ссылку на именованный элемент `<form>` с помощью такого выражения:

```
document.shipping_address
```

С помощью свойства `document.forms` обращение к форме, имеющей атрибут `name` (или `id`), можно записать более явно:

```
document.forms.shipping_address;
```

Объект `HTMLDocument` также определяет свойства-синонимы `embeds` и `plugins`, являющиеся коллекциями `HTMLCollection` элементов `<embed>`. Свойство `anchors` является нестандартным, но с его помощью можно получить доступ к элементам `<a>`, имеющим атрибут `name`, но не имеющим атрибут `href`. Свойство `scripts` определено стандартом HTML5 и является коллекцией `HTMLCollection` элементов `<script>`, но к моменту написания этих строк оно было реализовано не во всех броузерах.

Кроме того, объект `HTMLDocument` определяет два свойства, каждое из которых ссылается не на коллекцию, а на единственный элемент. Свойство `document.body` представляет элемент `<body>` HTML-документа, а свойство `document.head` – элемент `<head>`. Эти свойства всегда определены в документе: даже если в исходном документе отсутствуют элементы `<head>` и `<body>`, броузер создаст их неявно. Свойство `documentElement` объекта `Document` ссылается на корневой элемент документа. В HTML-документах он всегда представляет элемент `<html>`.

Объекты NodeList и HTMLCollection

Методы `getElementsByName()` и `getElementsByTagName()` возвращают объекты `NodeList`, а такие свойства, как `document.images` и `document.forms`, являются объектами `HTMLCollection`.

Эти объекты являются объектами, подобными массивам, доступным только для чтения (раздел 7.11). Они имеют свойство `length` и могут индексироваться (только для чтения) подобно настоящим массивам. Содержимое объекта `NodeList` или `HTMLCollection` можно обойти с помощью стандартного цикла, например:

```
for(var i = 0; i < document.images.length; i++)  // Обойти все изобр.
    document.images[i].style.display = "none";  // ...и скрыть их.
```

Для объектов `NodeList` и `HTMLCollection` нельзя непосредственно вызывать методы класса `Array`, но их можно вызывать косвенно:

```
var content = Array.prototype.map.call(
                    document.getElementsByTagName("p"),
                    function(e) { return e.innerHTML; });
```

Объекты `HTMLCollection` могут иметь дополнительные именованные свойства и могут индексироваться не только числами, но и строками.

По историческим причинам оба объекта, `NodeList` и `HTMLCollection`, могут также играть роль функций: вызов их с числовым или строковым аргументом равносилен операции индексирования числом или строкой. Однако использование этой причудливой особенности может сбивать с толку.

Интерфейсы обоих объектов, `NodeList` и `HTMLCollection`, проектировались под другие языки программирования, не такие динамические, как JavaScript. Оба определяют метод `item()`. Он принимает целое число и возвращает элемент с этим индексом. Однако в программах на языке JavaScript нет нужды использовать этот метод, так как можно использовать простую операцию индексирования массива. Аналогично `HTMLCollection` определяет метод `namedItem()`, возвращающий значение именованного свойства, но в программах на языке JavaScript вместо него можно использовать операции индексирования массива и обращения к свойствам.

Одна из наиболее важных и интересных особенностей объектов `NodeList` и `HTMLCollection` состоит в том, что они не являются статическими слепками документа, а продолжают «жить», и списки элементов, которые они представляют, изменяются по мере изменения документа. Если вызвать метод `getElementsByTagName('div')` для документа, в котором отсутствуют элементы `<div>`, он вернет объект `NodeList`, свойство `length` которого будет равно 0. Если затем вставить в документ новый элемент `<div>`, этот элемент автоматически станет членом коллекции `NodeList`, а ее свойство `length` станет равно 1.

Обычно такая динамичность элементов `NodeList` и `HTMLCollection` бывает весьма полезна. Однако если добавлять или удалять элементы из документа в процессе итераций по коллекции `NodeList`, потребуется предварительно создать статическую копию объекта `NodeList`:

```
var snapshot = Array.prototype.slice.call(nodelist, 0);
```

15.2.4. Выбор элементов по классу CSS

Значением HTML-атрибута `class` является список из нуля или более идентификаторов, разделенных пробелами. Он дает возможность определять множества связанных элементов документа: любые элементы, имеющие в атрибуте `class` один и тот же идентификатор, являются частью одного множества. Слово `class` зарезервировано в языке JavaScript, поэтому для хранения значения HTML-атрибута `class` в клиентском JavaScript используется свойство `className`. Обычно атрибут `class` используется вместе с каскадными таблицами стилей CSS, с целью применить общий стиль отображения ко всем членам множества, и мы еще будем рассматривать эту тему в главе 16. Однако кроме этого, стандарт HTML5 определяет метод `getElementsByClassName()`, позволяющий выбирать множества элементов документа на основе идентификаторов в их атрибутах `class`.

Подобно методу `getElementsByTagName()`, метод `getElementsByClassName()` может вызываться и для HTML-документов, и для HTML-элементов, и возвращает «живой» объект `NodeList`, содержащий все потомки документа или элемента, соответствующие критерию поиска. Метод `getElementsByClassName()` принимает единственный строковый аргумент, но в самой строке может быть указано несколько идентификаторов, разделенных пробелами. Соответствующими будут считаться все элементы, атрибуты `class` которых содержат все указанные идентификаторы. Порядок следования идентификаторов не имеет значения. Обратите внимание, что и в атрибуте `class`, и в аргументе метода `getElementsByClassName()` идентификаторы классов разделяются пробелами, а не запятыми. Ниже приводится несколько примеров использования метода `getElementsByClassName()`:

```
// Отыскать все элементы с идентификатором "warning" в атрибуте class
var warnings = document.getElementsByClassName("warning");

// Отыскать всех потомков элемента с именем "log" с идентификаторами "error"
// и "fatal" в атрибуте class
var log = document.getElementById("log");
var fatal = log.getElementsByClassName("fatal error");
```

Современные броузеры отображают HTML-документы в «режиме совместимости» или в «стандартном режиме», в зависимости от строгости объявления `<!DOCTYPE>` в начале документа. Режим совместимости поддерживается для сохранения обратной совместимости, и одна из его особенностей состоит в том, что идентификаторы классов в атрибуте `class` и каскадных таблицах стилей CSS нечувствительны к регистру символов. Метод `getElementsByClassName()` следует алгоритму сопоставления, используемому таблицами стилей. Если документ отображается в режиме совместимости, метод сравнивает строки без учета регистра символов. В противном случае сравнение выполняется с учетом регистра символов.

К моменту написания этих строк метод `getElementsByClassName()` был реализован во всех текущих броузерах, за исключением IE8 и более ранних версий. Однако IE8 поддерживает метод `querySelectorAll()`, описываемый в следующем разделе, на основе которого можно реализовать метод `getElementsByClassName()`.

15.2.5. Выбор элементов с использованием селекторов CSS

Каскадные таблицы стилей CSS имеют очень мощные синтаксические конструкции, известные как *селекторы*, позволяющие описывать элементы или множества элементов документа. Полное описание синтаксиса селекторов CSS выходит далеко за рамки этой книги[1], однако несколько примеров помогут прояснить их основы. Элементы можно описать с помощью имени тега и атрибутов `id` и `class`:

---

[1] Определение селекторов CSS3 можно найти по адресу *http://www.w3.org/TR/css3-selectors/*.

```
#nav        // Элемент с атрибутом id="nav"
div         // Любой элемент <div>
.warning    // Любой элемент с идентификатором "warning" в атрибуте class
```

В более общем случае элементы можно выбирать, опираясь на значения атрибутов:

```
p[lang="fr"] // Абзац с текстом на французском языке: <p lang="fr">
*[name="x"]  // Любой элемент с атрибутом name="x"
```

Эти простейшие селекторы можно комбинировать:

```
span.fatal.error        // Любой элемент <span> с классами "fatal" и "error"
span[lang="fr"].warning // Любое предупреждение на французском языке
```

С помощью селекторов можно также определять взаимоотношения между элементами:

```
#log span // Любой элемент <span>, являющийся потомком элемента с id="log"
#log>span // Любой элемент <span>, дочерний по отношению к элементу с id="log"
body>h1:first-child // Первый элемент <h1>, дочерний по отношению к <body>
```

Селекторы можно комбинировать для выбора нескольких элементов или множеств элементов:

```
div, #log // Все элементы <div> плюс элемент с id="log"
```

Как видите, селекторы CSS позволяют выбирать элементы всеми способами, описанными выше: по значению атрибута `id` и `name`, по имени тега и по имени класса. Наряду со стандартизацией селекторов CSS3, другой стандарт консорциума W3C, известный как «Selectors API» (API селекторов), определяет методы JavaScript для получения элементов, соответствующих указанному селектору.[1] Ключевым в этом API является метод `querySelectorAll()` объекта `Document`. Он принимает единственный строковый аргумент с селектором CSS и возвращает объект `NodeList`, представляющий все элементы документа, соответствующие селектору. В отличие от ранее описанных методов выбора элементов, объект `NodeList`, возвращаемый методом `querySelectorAll()`, не является «живым»: он хранит элементы, которые соответствовали селектору на момент вызова метода, и не отражает последующие изменения в документе. В случае отсутствия элементов, соответствующих селектору, метод `querySelectorAll()` вернет пустой `NodeList`. Если методу `querySelectorAll()` передать недопустимую строку, он возбудит исключение.

В дополнение к методу `querySelectorAll()` объект документа также определяет метод `querySelector()`, подобный методу `querySelectorAll()`, – с тем отличием, что он возвращает только первый (в порядке следования в документе) соответствующий элемент или `null`, в случае отсутствия соответствующих элементов.

Эти два метода также определяются классом `Elements` (и классом `DocumentFragment`, о котором рассказывается в разделе 15.6.4). Когда они вызываются относительно элемента, поиск соответствия заданному селектору выполняется во всем документе, а затем результат фильтруется так, чтобы в нем остались только потомки использованного элемента. Такой подход может показаться противоречащим здравому смыслу, так как он означает, что строка селектора может включать предков элемента, для которого выполняется сопоставление.

---

[1] Стандарт «Selectors API» не является частью стандарта HTML5, но тесно связан с ним. Подробности смотрите по адресу *http://www.w3.org/TR/selectors-api/*.

Обратите внимание, что стандарт CSS определяет псевдоэлементы `:first-line` и `:first-letter`. В CSS им соответствуют не фактические элементы, а части текстовых узлов. Они не будут обнаруживать совпадений, если использовать их вместе с методом `querySelectorAll()` или `querySelector()`. Кроме того, многие броузеры не возвращают результатов сопоставления с псевдоклассами `:link` и `:visited`, потому что в противном случае это позволило бы получить информацию об истории посещений страниц пользователем.

Методы `querySelector()` и `querySelectorAll()` поддерживают все текущие броузеры. Тем не менее обратите внимание, что спецификации этих методов не требуют поддержки селекторов CSS3: производителям броузеров предлагается реализовать поддержку того же набора селекторов, который поддерживается в каскадных таблицах стилей. Текущие броузеры, кроме IE, поддерживают селекторы CSS3. IE7 и 8 поддерживают селекторы CSS2. (Ожидается, что IE9 будет поддерживать CSS3.)

Метод `querySelectorAll()` является идеальным инструментом выбора элементов: это очень мощный механизм, с помощью которого клиентские программы на языке JavaScript могут выбирать элементы документа для выполнения операций над ними. К счастью, селекторы CSS можно использовать даже в броузерах, не имеющих собственной поддержки метода `querySelectorAll()`. Похожий механизм запросов на основе селекторов в библиотеке jQuery (глава 19) является центральной парадигмой программирования. Веб-приложения на основе jQuery могут использовать переносимый, совместимый с разными типами броузеров эквивалент метода `querySelectorAll()`, который называется `$()`.

Программный код, выполняющий в библиотеке jQuery сопоставление с селекторами CSS, был реструктурирован и вынесен в самостоятельную библиотеку с именем Sizzle, которая была заимствована фреймворком Dojo и другими клиентскими библиотеками.[1] Преимущество использования библиотек, таких как Sizzle (или библиотек, использующих Sizzle), в том, что выбор элементов можно производить даже в старых броузерах, и при этом обеспечивается поддержка базового набора селекторов, которые гарантированно будут работать во всех броузерах.

15.2.6. document.all[]

До того, как модель DOM была стандартизована, в IE4 была реализована коллекция `document.all[]`, представляющая все элементы (кроме текстовых узлов `Text`) в документе. Впоследствии коллекцию `document.all[]` заменили стандартные методы, такие как `getElementById()` и `getElementsByTagName()`, и теперь она считается устаревшей и не должна использоваться. Однако в свое время появление этой коллекции произвело целую революцию, и даже сейчас все еще можно встретить сценарии, использующие ее следующими способами:

```
document.all[0]           // Первый элемент документа
document.all["navbar"]    // Элемент (или элементы) со значением "navbar"
                          // в атрибуте id или name
document.all.navbar       // То же самое
document.all.tags("div")  // Все элементы <div> в документе
document.all.tags("p")[0] // Первый элемент <p> в документе
```

---

[1] Самостоятельная версия библиотеки Sizzle доступна по адресу *http://sizzlejs.com*.

15.3. Структура документа и навигация по документу

После выбора элемента документа иногда бывает необходимо отыскать структурно связанные части документа (родитель, братья, дочерний элемент). Объект `Document` можно представить как дерево объектов `Node`, как изображено на рис. 15.1. Тип `Node` определяет свойства, позволяющие перемещаться по такому дереву, которые будут рассматриваться в разделе 15.3.1. Существует еще один прикладной интерфейс навигации по документу, как дерева объектов `Element`. Этот более новый (и часто более простой в использовании) прикладной интерфейс рассматривается в разделе 15.3.2.

15.3.1. Документы как деревья узлов

Объект `Document`, его объекты `Element` и объекты `Text`, представляющие текстовые фрагменты в документе, – все они являются объектами `Node`. Класс `Node` определяет следующие важные свойства:

`parentNode`

Родительский узел данного узла или `null` для узлов, не имеющих родителя, таких как `Document`.

`childNodes`

Доступный для чтения объект, подобный массиву (`NodeList`), обеспечивающий «живое» представление дочерних узлов.

`firstChild`, `lastChild`

Первый и последний дочерние узлы или `null`, если данный узел не имеет дочерних узлов.

`nextSibling`, `previousSibling`

Следующий и предыдущий братские узлы. Братскими называются два узла, имеющие одного и того же родителя. Порядок их следования соответствует порядку следования в документе. Эти свойства связывают узлы в двусвязный список.

`nodeType`

Тип данного узла. Узлы типа `Document` имеют значение 9 в этом свойстве. Узлы типа `Element` – значение 1. Текстовые узлы типа `Text` – значение 3. Узлы типа `Comments` – значение 8 и узлы типа `DocumentFragment` – значение 11.

`nodeValue`

Текстовое содержимое узлов `Text` и `Comment`.

`nodeName`

Имя тега элемента `Element`, в котором все символы преобразованы в верхний регистр.

С помощью этих свойств класса `Node` можно сослаться на второй дочерний узел первого дочернего узла объекта `Document`, как показано ниже:

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Допустим, что рассматриваемый документ имеет следующий вид:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Тогда вторым дочерним узлом первого дочернего узла будет элемент `<body>`. В свойстве `nodeType` он содержит значение 1 и в свойстве `nodeName` – значение «BODY».

Однако, обратите внимание, что этот прикладной интерфейс чрезвычайно чувствителен к изменениям в тексте документа. Например, если в этот документ добавить единственный перевод строки между тегами `<html>` и `<head>`, этот символ перевода строки станет первым дочерним узлом (текстовым узлом `Text`) первого дочернего узла, а вторым дочерним узлом станет элемент `<head>`, а не `<body>`.

15.3.2. Документы как деревья элементов

Когда основной интерес представляют сами элементы документа, а не текст в них (и пробельные символы между ними), гораздо удобнее использовать прикладной интерфейс, позволяющий интерпретировать документ как дерево объектов `Element`, игнорируя узлы `Text` и `Comment`, которые также являются частью документа.

Первой частью этого прикладного интерфейса является свойство `children` объектов `Element`. Подобно свойству `childNodes`, его значением является объект `NodeList`. Однако, в отличие от свойства `childNodes`, список `children` содержит только объекты `Element`. Свойство `children` – нестандартное свойство, но оно реализовано во всех текущих броузерах. В IE это свойство было реализовано уже очень давно, и большинство других броузеров последовали его примеру. Последним основным броузером, реализовавшим его, стал Firefox 3.5.

Обратите внимание, что узлы `Text` и `Comment` не имеют дочерних узлов. Это означает, что описанное выше свойство `Node.parentNode` никогда не возвращает узлы типа `Text` или `Comment`. Значением свойства `parentNode` любого объекта `Element` всегда будет другой объект `Element` или корень дерева – объект `Document` или `DocumentFragment`.

Второй частью прикладного интерфейса навигации по элементам документа являются свойства объекта `Element`, аналогичные свойствам доступа к дочерним и братским узлам объекта `Node`:

`firstElementChild`, `lastElementChild`

Похожи на свойства `firstChild` и `lastChild`, но возвращают дочерние элементы.

`nextElementSibling`, `previousElementSibling`

Похожи на свойства `nextSibling` и `previousSibling`, но возвращают братские элементы.

`childElementCount`

Количество дочерних элементов. Возвращает то же значение, что и свойство `children.length`.

Эти свойства доступа к дочерним и братским элементам стандартизованы и реализованы во всех текущих броузерах, кроме IE.[1]

Поскольку прикладной интерфейс навигации по элементам документа реализован не во всех броузерах, вам может потребоваться определить переносимые функции навигации, как в примере 15.2.

---

[1] *http://www.w3.org/TR/ElementTraversal.*

Пример 15.2. Переносимые функции навигации по документу

```
/**
 * Возвращает ссылку на n-го предка элемента e или null, если нет такого предка
 * или если этот предок не является элементом Element
 * (например, Document или DocumentFragment).
 * Если в аргументе n передать 0, функция вернет сам элемент e.
 * Если в аргументе n передать 1 (или вообще опустить этот аргумент),
 * функция вернет родительский элемент.
 * Если в аргументе n передать 2, функция вернет родителя родительского элемента и т. д.
 */
function parent(e, n) {
    if (n === undefined) n = 1;
    while(n-- && e) e = e.parentNode;
    if (!e || e.nodeType !== 1) return null;
    return e;
}

/**
 * Возвращает n-й братский элемент элемента e.
 * Если в аргументе n передать положительное число, функция вернет следующий
 * n-й братский элемент.
 * Если в аргументе n передать отрицательное число, функция вернет предыдущий
 * n-й братский элемент.
 * Если в аргументе n передать ноль, функция вернет сам элемент e.
 */
function sibling(e,n) {
    while(e && n !== 0) {       // Если e не определен, просто вернуть его
        if (n > 0) {            // Отыскать следующий братский элемент
            if (e.nextElementSibling) e = e.nextElementSibling;
            else {
                for(e=e.nextSibling; e && e.nodeType !== 1; e=e.nextSibling)
                    /* пустой цикл */ ;
            }
            n--;
        }
        else {                  // Отыскать предыдущий братский элемент
            if (e.previousElementSibing) e = e.previousElementSibling;
            else {
                for(e=e.previousSibling;e&&e.nodeType!==1;e=e.previousSibling)
                    /* пустой цикл */ ;
            }
            n++;
        }
    }
    return e;
}

/**
 * Возвращает n-й дочерний элемент элемента e или null, если нет такого
 * дочернего элемента.
 * Если в аргументе n передать отрицательное число, поиск дочернего элемента
 * будет выполняться с конца. 0 соответствует первому дочернему элементу,
 * но -1 - последнему, -2 - второму с конца и т. д.
 */
```

```
function child(e, n) {
    if (e.children) {                       // Если массив children существует
        if (n < 0) n += e.children.length;  // Преобразовать отрицательное
                                            // число в индекс массива
        if (n < 0) return null;             // Если получилось отрицательное число,
                                            // значит, нет такого дочернего элемента
        return e.children[n];               // Вернуть заданный дочерний элемент
    }

    // Если элемент e не имеет массива children, начать поиск с первого
    // дочернего элемента, двигаясь вперед, или начать поиск с последнего
    // дочернего элемента, двигаясь назад.
    if (n >= 0) { // n - положительное: двигаться вперед, начиная с первого
        // Найти первый дочерний элемент элемента e
        if (e.firstElementChild) e = e.firstElementChild;
        else {
            for(e = e.firstChild; e && e.nodeType !== 1; e = e.nextSibling)
                /* пустой цикл */;
        }
        return sibling(e, n); // Вернуть n-го брата первого дочернего элемента
    }
    else { // n - отрицательное: двигаться назад, начиная с последнего
        if (e.lastElementChild) e = e.lastElementChild;
        else {
            for(e = e.lastChild; e && e.nodeType !== 1; e=e.previousSibling)
                /* пустой цикл */;
        }
        return sibling(e, n+1); // +1, чтобы преобразовать номер -1 дочернего
                                // в номер 0 братского для последнего
    }
}
```

Определение собственных методов элементов

Все текущие броузеры (включая IE8 и выше) реализуют модель DOM таким образом, что такие типы, как `Element` и `HTMLDocument`[1], являются классами, такими же как классы `String` и `Array`. Они не имеют конструкторов (как создавать новые объекты `Element`, будет показано далее в этой главе), но они имеют объекты-прототипы, которые вы можете расширять своими методами:

```
Element.prototype.next = function() {
    if (this.nextElementSibling) return this.nextElementSibling;
    var sib = this.nextSibling;
    while(sib && sib.nodeType !== 1) sib = sib.nextSibling;
    return sib;
};
```

---

[1] **IE8 поддерживает возможность расширения прототипов объектов `Element`, `HTMLDocument` и `Text`, но не поддерживает для объектов `Node`, `Document`, `HTMLElement` и всех подтипов типа `HTMLElement`.**

Функции, представленные в примере 15.2, не были реализованы в виде методов объекта Element лишь по той причине, что такая возможность не поддерживается в IE7.

Однако возможность расширения типов DOM может пригодиться для реализации особенностей, характерных для IE, в других броузерах. Как отмечалось выше, нестандартное свойство children объекта Element было впервые реализовано в IE и только потом – в других броузерах. Используя следующий программный код, можно реализовать это свойство в броузерах, не поддерживающих его, таких как Firefox 3.0:

```
// Реализация свойства Element.children в броузерах,
// не поддерживающих его
// Обратите внимание, что этот метод возвращает статический
// массив, а не "живой" NodeList
if (!document.documentElement.children) {
    Element.prototype.__defineGetter__("children", function() {
        var kids = [];
        for(var c = this.firstChild; c != null; c = c.nextSibling)
            if (c.nodeType === 1) kids.push(c);
        return kids;
    });
}
```

Метод __defineGetter__ (о нем рассказывается в разделе 6.7.1) не является стандартным, но его вполне можно использовать для обеспечения переносимости в таком программном коде, как этот.

15.4. Атрибуты

HTML-элементы состоят из имени тега и множества пар имя/значение, известных как атрибуты. Например, элемент <a>, определяющий гиперссылку, в качестве адреса назначения ссылки использует значение атрибута href. Значения атрибутов HTML-элементов доступны в виде свойств объектов HTMLElement, представляющих эти элементы. Кроме того, модель DOM определяет и другие механизмы получения и изменения значений XML-атрибутов и нестандартных HTML-атрибутов. Подробнее об этом рассказывается в следующих подразделах.

15.4.1. HTML-атрибуты как свойства объектов Element

Объекты HTMLElement, представляющие элементы HTML-документа, определяют свойства, доступные для чтения/записи, соответствующие HTML-атрибутам элементов. Объект HTMLElement определяет свойства для поддержки универсальных HTTP-атрибутов, таких как id, title, lang и dir, и даже свойства-обработчики событий, такие как onclick. Специализированные подклассы класса Element определяют атрибуты, характерные для представляемых ими элементов. Например, узнать URL-адрес изображения можно, обратившись к свойству src объекта HTMLElement, представляющего элемент <img>:

```
var image = document.getElementById("myimage");
var imgurl = image.src; // Атрибут src определяет URL-адрес изображения
image.id === "myimage"  // Потому что поиск элемента выполнялся по id
```

Аналогично можно устанавливать атрибуты элемента `<form>`, определяющие порядок отправки формы:

```
var f = document.forms[0];                      // Первый элемент <form> в документе
f.action = "http://www.example.com/submit.php"; // Установить URL отправки.
f.method = "POST";                              // Тип HTTP-запроса
```

Имена атрибутов в разметке HTML не чувствительны к регистру символов, в отличие от имен свойств в языке JavaScript. Чтобы преобразовать имя атрибута в имя свойства в языке JavaScript, его нужно записать символами в нижнем регистре. Однако, если имя атрибута состоит из более чем одного слова, первый символ каждого слова, кроме первого, записывается в верхнем регистре, например: `defaultChecked` и `tabIndex`.

Имена некоторых HTML-атрибутов совпадают с зарезервированными словами языка JavaScript. Имена свойств, соответствующих таким атрибутам, начинаются с приставки «html». Например, HTML-атрибуту `for` (элемента `<label>`) в языке JavaScript соответствует свойство с именем `htmlFor`. Очень важный HTML-атрибут `class`, имя которого совпадает с зарезервированным (но не используемым) в языке JavaScript словом «class», является исключением из этого правила: в программном коде на языке JavaScript ему соответствует свойство `className`. Мы еще встретимся со свойством `className` в главе 16.

Свойства, представляющие HTML-атрибуты, обычно имеют строковые значения. Если атрибут имеет логическое или числовое значение (например, атрибуты `defaultChecked` и `maxLength` элемента `<input>`), значением соответствующего свойства будет логическое или числовое значение, а не строка. Значениями атрибутов обработчиков событий всегда являются объекты `Function` (или `null`). Спецификация HTML5 определяет несколько атрибутов (таких как атрибут `form` элемента `<input>` и родственных ему элементов), которые преобразуются в фактические объекты `Element`. Наконец, значением свойства `style` любого HTML-элемента является объект `CSSStyleDeclaration`, а не строка. Поближе с этим важным свойством мы познакомимся в главе 16.

Обратите внимание, что основанный на свойствах прикладной интерфейс получения доступа к значениям атрибутов не позволяет удалять атрибуты из элементов. В частности, для этих целей нельзя использовать оператор `delete`. Для этой цели можно использовать прием, который описывается в следующем разделе.

15.4.2. Доступ к нестандартным HTML-атрибутам

Как описывалось выше, тип `HTMLElement` и его подтипы определяют свойства, соответствующие стандартным атрибутам HTML-элементов. Однако тип `Element` определяет дополнительные методы `getAttribute()` и `setAttribute()`, которые можно использовать для доступа к нестандартным HTML-атрибутам, а также обращаться к атрибутам элементов XML-документа:

```
var image = document.images[0];
var width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

Пример выше демонстрирует два важных отличия между этими методами и прикладным интерфейсом, основанным на свойствах и описанным выше. Во-первых, значения всех атрибутов они интерпретируют как строки. Метод `getAttribute()` никогда не вернет число, логическое значение или объект. Во-вторых, эти методы принимают стандартные имена атрибутов, даже если они совпадают с зарезервированными словами языка JavaScript. Имена атрибутов HTML-элементов нечувствительны к регистру символов.

Класс `Element` также определяет два родственных метода, `hasAttribute()` и `removeAttribute()`. Первый из них проверяет присутствие атрибута с указанным именем, а второй удаляет атрибут. Эти методы особенно удобны при работе с логическими атрибутами: для этих атрибутов (таких как атрибут `disabled` HTML-форм) важно их наличие или отсутствие в элементе, а не их значения.

Если вам приходится работать с XML-документами, содержащими атрибуты из других пространств имен, вы можете использовать варианты этих четырех методов, позволяющие указывать имя пространства имен: `getAttributeNS()`, `setAttributeNS()`, `hasAttributeNS()` и `removeAttributeNS()`. Вместо единственного строкового аргумента с именем атрибута эти методы принимают два аргумента. В первом передается URI-идентификатор, определяющий пространство имен. Во втором аргументе обычно передается неквалифицированное локальное имя атрибута из этого пространства имен. Исключением является метод `setAttributeNS()`, которому во втором атрибуте необходимо передавать квалифицированное имя атрибута, включающее идентификатор пространства имен. Более полная информация об этих методах доступа к атрибутам из других пространств имен приводится в четвертой части книги.

15.4.3. Атрибуты с данными

Иногда бывает желательно добавить в HTML-элементы дополнительные данные, обычно когда предусматривается возможность выбора этих элементов в JavaScript-сценариях и выполнения некоторых операций с ними. Иногда это можно реализовать, добавив специальные идентификаторы в атрибут `class`. Иногда, когда речь заходит о более сложных данных, программисты прибегают к использованию нестандартных атрибутов. Как отмечалось выше, для чтения и изменения значений нестандартных атрибутов можно использовать методы `getAttribute()` и `setAttribute()`. Платой за это будет несоответствие документа стандарту.

Стандарт HTML5 предоставляет решение этой проблемы. В документах, соответствующих стандарту HTML5, все атрибуты, имена которых состоят только из символов в нижнем регистре и начинаются с приставки «data-», считаются допустимыми. Эти «атрибуты с данными» не оказывают влияния на представление элементов, в которых присутствуют, и обеспечивают стандартный способ включения дополнительных данных без нарушения стандартов.

Кроме того, стандарт HTML5 определяет в объекте `Element` свойство `dataset`. Это свойство ссылается на объект, который имеет свойства, имена которых соответствуют именам атрибутов `data-` без приставки. То есть свойство `dataset.x` будет хранить значение атрибута `data-x`. Имена атрибутов с дефисами отображаются в имена свойств с переменным регистром символов: атрибут `data-jquery-test` превратится в свойство `dataset.jqueryTest`.

Ниже приводится более конкретный пример. Допустим, что в документе имеется следующий фрагмент разметки:

```
<span class="sparkline" data-ymin="0" data-ymax="10">
1 1 1 2 2 3 4 5 5 4 3 5 6 7 7 4 2 1
</span>
```

Sparkline – это маленькое изображение, обычно некоторый график, предназначенное для отображения в потоке текста. Чтобы сгенерировать такое изображение, необходимо извлечь значение атрибута с данными, как показано ниже:

```
// Предполагается, что в броузере поддерживается метод Array.map(),
// определяемый стандартом ES5 (или реализована его имитация)
var sparklines = document.getElementsByClassName("sparkline");
for(var i = 0; i < sparklines.length; i++) {
    var dataset = sparklines[i].dataset;
    var ymin = parseFloat(dataset.ymin);
    var ymax = parseFloat(dataset.ymax);
    var data = sparklines[i].textContent.split(" ").map(parseFloat);
    drawSparkline(sparklines[i], ymin, ymax, data); // Еще не реализована
}
```

На момент написания этих строк свойство `dataset` еще не было реализовано в текущих броузерах, поэтому представленное выше решение можно было бы реализовать так:

```
var sparklines = document.getElementsByClassName("sparkline");
    for(var i = 0; i < sparklines.length; i++) {
        var elt = sparklines[i];
        var ymin = parseFloat(elt.getAttribute("data-ymin"));
        var ymin = parseFloat(elt.getAttribute("data-ymax"));
        var points = elt.getAttribute("data-points");
        var data = elt.textContent.split(" ").map(parseFloat);
        drawSparkline(elt, ymin, ymax, data); // Еще не реализована
}
```

Обратите внимание, что свойство `dataset` является (или будет, когда будет реализовано) «живым», двунаправленным интерфейсом к атрибутам `data-` элемента. Изменение или удаление свойства объекта `dataset` приводит к изменению или удалению соответствующего атрибута `data-` элемента.

Функция `drawSparkline()` в примере выше является вымышленной, однако в примере 21.13 демонстрируется прием рисования внутристрочных диаграмм (sparklines), подобных диаграмме в примере выше, с использованием элемента `<canvas>`.

15.4.4. Атрибуты как узлы типа Attr

Существует несколько способов работы с атрибутами элементов. Тип `Node` определяет свойство `attributes`. Это свойство имеет значение `null` для всех узлов, не являющихся объектами `Element`. Свойство `attributes` объектов `Element` является объектом, подобным массиву, доступным только для чтения, представляющим все атрибуты элемента. Подобно спискам `NodeList`, объект `attributes` не является статической копией. Он может индексироваться числами, что означает возможность перечисления всех атрибутов элемента, а также именами атрибутов:

```
document.body.attributes[0]       // Первый атрибут элемента <body>
document.body.attributes.bgcolor  // Атрибут bgcolor элемента <body>
document.body.attributes["ONLOAD"] // Атрибут onload элемента <body>
```

Значениями, получаемыми в результате индексирования объекта `attributes`, являются объекты `Attr`. Объекты `Attr` – это специализированный подтип `Node`, но в действительности никогда не используемые в таком качестве. Свойства `name` и `value` объектов `Attr` возвращают имя и значение атрибута.

15.5. Содержимое элемента

Взгляните еще раз на рис. 15.1 и попробуйте ответить на вопрос: какой объект представляет «содержимое» элемента `<p>`. На этот вопрос можно дать три ответа:

- Содержимым является строка разметки HTML «This is a <i>simple</i> document».
- Содержимым является простая текстовая строка «This is a simple document».
- Содержимым является узел типа `Text`, узел типа `Element`, включающий дочерний узел `Text`, и еще один узел типа `Text`.

Все три ответа являются верными, и каждый ответ ценен по-своему. В следующих разделах описывается, как работать с представлением в виде разметки HTML, с представлением в виде простого текста и с представлением в виде дерева объектов.

15.5.1. Содержимое элемента в виде HTML

При чтении свойства `innerHTML` объекта `Element` возвращается содержимое этого элемента в виде строки разметки. Попытка изменить значение этого свойства приводит к вызову синтаксического анализатора веб-броузера и замещению текущего содержимого элемента разобранным представлением новой строки. (Несмотря на свое название, свойство `innerHTML` может использоваться для работы не только с HTML-, но и с XML-элементами.)

Веб-броузеры прекрасно справляются с синтаксическим анализом разметки HTML, поэтому операция изменения значения свойства `innerHTML` обычно достаточно эффективна, несмотря на необходимость синтаксического анализа. Тем не менее обратите внимание, что многократное добавление фрагментов текста в свойство `innerHTML` с помощью оператора += обычно далеко не эффективное решение, потому что требует выполнения двух шагов – сериализации и синтаксического анализа.

Впервые свойство `innerHTML` было реализовано в IE4. Несмотря на то что оно достаточно давно поддерживается всеми броузерами, это свойство было стандартизовано только с появлением стандарта HTML5. Спецификация HTML5 требует, чтобы свойство `innerHTML` было реализовано не только в объекте `Element`, но и в объекте `Document`, однако пока этому требованию отвечают не все броузеры.

Кроме того, спецификация HTML5 стандартизует свойство с именем `outerHTML`. При обращении к свойству `outerHTML` оно возвращает строку разметки HTML или XML, содержащую открывающий и закрывающий теги элемента, которому принадлежит это свойство. При записи нового значения в свойство `outerHTML` элемента новое содержимое замещает элемент целиком. Свойство outerHTML определено только для узлов типа Element, оно отсутствует в объекте Document. К моменту написания этих строк свойство outerHTML поддерживалось всеми текущими броузерами, кроме Firefox. (В примере 15.5, далее в этой главе, приводится реализация свойства outerHTML на основе свойства innerHTML.)

Еще одной особенностью, впервые появившейся в IE и стандартизованной спецификацией HTML5, является метод insertAdjacentHTML(), дающий возможность вставить строку с произвольной разметкой HTML, прилегающую («adjacent») к указанному элементу. Разметка передается методу во втором аргументе, а точное значение слова «прилегающая» («adjacent») зависит от значения первого аргумента. Этот первый аргумент должен быть строкой с одним из значений: «beforebegin», «afterbegin», «beforeend» или «afterend». Эти значения определяют позицию вставки, как изображено на рис. 15.3.

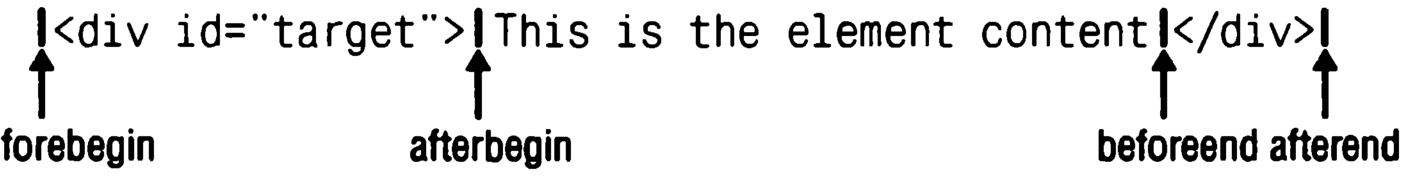


Рис. 15.3. Позиции вставки в вызове метода insertAdjacentHTML()

Метод insertAdjacentHTML() не поддерживается текущей версией Firefox. Далее в этой главе будет представлен пример 15.6, демонстрирующий, как можно реализовать метод insertAdjacentHTML() с применением свойства innerHTML и как можно написать методы вставки разметки HTML, не требующие указывать позицию вставки с помощью строкового аргумента.

15.5.2. Содержимое элемента в виде простого текста

Иногда бывает необходимо получить содержимое элемента в виде простого текста или вставить простой текст в документ (без необходимости экранировать угловые скобки и амперсанды, используемые в разметке HTML). Стандартный способ выполнения этих операций основан на использовании свойства textContent объекта Node:

```
var para = document.getElementsByTagName("p")[0]; // Первый <p> в документе
var text = para.textContent;                      // Текст "This is a simple document."
para.textContent = "Hello World!";                // Изменит содержимое абзаца
```

Свойство textContent поддерживается всеми текущими броузерами, кроме IE. В IE вместо него можно использовать свойство innerText. Впервые свойство innerText было реализовано в IE4 и поддерживается всеми текущими броузерами, кроме Firefox.

Свойства textContent и innerText настолько похожи, что обычно могут быть взаимозаменяемы при использовании. Однако будьте внимательны и отличайте пустые элементы (строка "" в языке JavaScript интерпретируется как ложное значение) от неопределенных свойств:

```
/**
 * При вызове с одним аргументом возвращает значение свойства textContent
 * или innerText элемента. При вызове с двумя аргументами записывает
 * указанное значение в свойство textContent или innerText элемента.
 */
function textContent(element, value) {
    var content = element.textContent;     // Свойство textContent определено?
    if (value === undefined) {             // Если аргумент value не указан,
        if (content !== undefined) return content; // вернуть текущий текст
        else return element.innerText;
    }
    else {                                 // Иначе записать текст
        if (content !== undefined) element.textContent = value;
        else element.innerText = value;
    }
}
```

Свойство `textContent` возвращает результат простой конкатенации всех узлов `Text`, потомков указанного элемента. Свойство `innerText` не обладает четко определенным поведением и имеет несколько отличий от свойства `textContent`. `innerText` не возвращает содержимое элементов `<script>`. Из возвращаемого им текста удаляются лишние пробелы и предпринимается попытка сохранить табличное форматирование. Кроме того, для некоторых элементов таблиц, таких как `<table>`, `<tbody>` и `<tr>`, свойство `innerText` доступно только для чтения.

Текст в элементах <script>

Встроенные элементы `<script>` (т. е. без атрибута `src`) имеют свойство `text`, которое можно использовать для получения их содержимого в виде текста. Содержимое элементов `<script>` никогда не отображается в броузерах, а HTML-парсеры игнорируют угловые скобки и амперсанды внутри сценариев. Это делает элемент `<script>` идеальным средством встраивания произвольных текстовых данных, доступных для использования веб-приложением. Достаточно просто определить в атрибуте `type` элемента какое-либо значение (такое как «text/x-custom-data»), чтобы сообщить, что этот сценарий не содержит выполняемый программный код на языке JavaScript. В этом случае интерпретатор JavaScript будет игнорировать сценарий, но сам элемент будет включен в дерево документа, а содержащиеся в нем данные можно будет получить с помощью свойства `text`.

15.5.3. Содержимое элемента в виде текстовых узлов

Еще одним средством доступа к содержимому элемента является список дочерних узлов, каждый из которых может иметь свое множество дочерних узлов. Когда речь заходит о содержимом элемента, наибольший интерес обычно представляют текстовые узлы. При работе с XML-документами необходимо также быть готовыми встретить узлы `CDATASection` – подтип класса `Text` – представляющие содержимое разделов `CDATA`.

Пример 15.3 демонстрирует функцию `textContent()`, которая выполняет рекурсивный обход дочерних элементов и объединяет текст, содержащийся во всех тек-

стовых узлах-потомках. Чтобы было более понятно, напомню, что свойство nodeValue (определяемое типом Node) хранит содержимое текстового узла.

Пример 15.3. Поиск всех текстовых узлов, потомков указанного элемента

```
// Возвращает простое текстовое содержимое элемента e, выполняя рекурсивный
// обход всех дочерних элементов. Этот метод действует подобно свойству textContent
function textContent(e) {
    var child, type, s = "";          // s хранит текст всех дочерних узлов
    for(child = e.firstChild; child != null; child = child.nextSibling) {
        type = child.nodeType;
        if (type === 3 || type === 4)  // Узлы типов Text и CDATASection
            s += child.nodeValue;
        else if (type === 1)           // Рекурсивный обход узлов типа Element
            s += textContent(child);
    }
    return s;
}
```

Свойство nodeValue доступно для чтения и записи, и с его помощью можно изменять содержимое в отображаемых узлах Text и CDATASection. Оба типа, Text и CDATASection, являются подтипами класса CharacterData, описание которого приводится в четвертой части книги. Класс CharacterData определяет свойство data, которое хранит тот же текст, что и свойство nodeValue. Следующая функция преобразует символы текстового содержимого узлов типа Text в верхний регистр, устанавливая значение свойства data:

```
// Рекурсивно преобразует символы всех текстовых узлов-потомков
// элемента n в верхний регистр.
function upcase(n) {
    if (n.nodeType == 3 || n.nodeTyep == 4) // Если n - объект Text или CDATA
        n.data = n.data.toUpperCase();      // преобразовать в верхний регистр
    else                                    // Иначе рекурсия по дочерним узлам
        for(var i = 0; i < n.childNodes.length; i++)
            upcase(n.childNodes[i]);
}
```

Класс CharacterData также определяет редко используемые методы добавления в конец, удаления, вставки и замены текста в узлах Text или CDATASection. Кроме изменения содержимого имеющихся текстовых узлов этот класс позволяет также вставлять в элементы Element новые текстовые узлы или замещать существующие текстовые узлы новыми. Создание, вставка и удаление узлов – тема следующего раздела.

15.6. Создание, вставка и удаление узлов

Мы уже знаем, как получать и изменять содержимое документа, используя строки с разметкой HTML и с простым текстом. Мы также знаем, как выполнять обход документа для исследования отдельных узлов Element и Text, составляющих его содержимое. Однако точно так же существует возможность изменения документа на уровне отдельных узлов. Тип Document определяет методы создания объектов Element и Text, а тип Node определяет методы для вставки, удаления и замены узлов в дереве. Приемы создания и вставки узлов уже были показаны в примере 13.4, который повторяется ниже:

```
// Асинхронная загрузка сценария из указанного URL-адреса и его выполнение
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; // Отыскать <head>
    var s = document.createElement("script");            // Создать элемент <script>
    s.src = url;                                         // Установить его атрибут src
    head.appendChild(s);                                 // Вставить <script> в <head>
}
```

В следующих подразделах более подробно и с примерами рассказывается о создании новых узлов, о вставке и удалении узлов, а также об использовании объектов `DocumentFragment`, упрощающих работу с множеством узлов.

15.6.1. Создание узлов

Как было показано в примере выше, создавать новые узлы `Element` можно с помощью метода `createElement()` объекта `Document`. Этому методу необходимо передать имя тега: это имя не чувствительно к регистру символов при работе с HTML-документами и чувствительно при работе с XML-документами.

Для создания текстовых узлов существует аналогичный метод:

```
var newnode = document.createTextNode("содержимое текстового узла");
```

Кроме того, объект `Document` определяет и другие фабричные методы, такие как редко используемый метод `createComment()`. Один такой метод, `createDocumentFragment()`, мы будем использовать в разделе 15.6.4. При работе с документами, в которых используются пространства имен XML, можно использовать метод `createElementNS()`, позволяющий указывать URI-идентификатор пространства имен и имя тега создаваемого объекта `Element`.

Еще один способ создания в документе новых узлов заключается в копировании существующих узлов. Каждый узел имеет метод `cloneNode()`, возвращающий новую копию узла. Если передать ему аргумент со значением `true`, он рекурсивно создаст копии всех потомков, в противном случае будет создана лишь поверхностная копия. В броузерах, отличных от IE, объект `Document` дополнительно определяет похожий метод с именем `importNode()`. Если передать ему узел из другого документа, он вернет копию, пригодную для вставки в текущий документ. Если передать ему значение `true` во втором аргументе, он рекурсивно импортирует все узлы-потомки.

15.6.2. Вставка узлов

После создания нового узла его можно вставить в документ с помощью методов типа `Node`: `appendChild()` или `insertBefore()`. Метод `appendChild()` вызывается относительно узла `Element`, в который требуется вставить новый узел, и вставляет указанный узел так, что тот становится последним дочерним узлом (значением свойства `lastChild`).

Метод `insertBefore()` похож на метод `appendChild()`, но он принимает два аргумента. В первом аргументе указывается вставляемый узел, а во втором – узел, перед которым должен быть вставлен новый узел. Этот метод вызывается относительно объекта узла, который станет родителем нового узла, а во втором аргументе должен передаваться дочерний узел этого родителя. Если во втором аргументе передать null, метод insertBefore() будет вести себя, как appendChild(), и вставит узел в конец.

Ниже приводится простая функция вставки узла в позицию с указанным числовым индексом. Она демонстрирует применение обоих методов, appendChild() и insertBefore():

```
// Вставляет узел child в узел parent так, что он становится n-м дочерним узлом
function insertAt(parent, child, n) {
    if (n < 0 || n > parent.childNodes.length)
        throw new Error("недопустимый индекс");
    else if (n == parent.childNodes.length) parent.appendChild(child);
    else parent.insertBefore(child, parent.childNodes[n]);
}
```

Если метод appendChild() или insertBefore() используется для вставки узла, который уже находится в составе документа, этот узел автоматически будет удален из текущей позиции и вставлен в новую позицию; нет необходимости явно удалять узел. Пример 15.4 демонстрирует функцию сортировки строк таблицы по значениям ячеек в указанном столбце. Она не создает новые узлы и для изменения порядка следования существующих узлов использует метод appendChild().

Пример 15.4. Сортировка строк таблицы

```
// Сортирует строки в первом элементе <tbody> указанной таблицы по значениям
// n-й ячейки в каждой строке. Использует функцию сравнения, если она указана.
// Иначе сортировка выполняется в алфавитном порядке.
function sortrows(table, n, comparator) {
    var tbody = table.tBodies[0]; // Первый <tbody>; возможно созданный неявно
    var rows = tbody.getElementsByTagName("tr"); // Все строки в tbody
    rows = Array.prototype.slice.call(rows,0);   // Скопировать в массив

    // Отсортировать строки по содержимому n-го элемента <td>
    rows.sort(function(row1,row2) {
        var cell1 = row1.getElementsByTagName("td")[n];  // n-е ячейки
        var cell2 = row2.getElementsByTagName("td")[n];  // двух строк
        var val1 = cell1.textContent || cell1.innerText; // текстовое содерж.
        var val2 = cell2.textContent || cell2.innerText; // двух ячеек
        if (comparator) return comparator(val1, val2);   // Сравнить!
        if (val1 < val2) return -1;
        else if (val1 > val2) return 1;
        else return 0;
    });

    // Добавить строки в конец tbody в отсортированном порядке.
    // При этом строки автоматически будут удалены из их текущих позиций,
    // благодаря чему отпадает необходимость явно удалять их. Если <tbody> содержит
    // какие-либо другие узлы, отличные от элементов <tr>, эти узлы "всплывут" наверх.
    for(var i = 0; i < rows.length; i++) tbody.appendChild(rows[i]);
}

// Отыскивает в таблице элементы <th> (предполагается, что в таблице существует
// только одна строка с ними) и добавляет в них возможность обработки щелчка мышью,
// чтобы щелчок на заголовке столбца вызывал сортировку таблицы по этому столбцу.
```

```
function makeSortable(table) {
    var headers = table.getElementsByTagName("th");
    for(var i = 0; i < headers.length; i++) {
        (function(n) { // Чтобы создать локальную область видимости
            headers[i].onclick = function() { sortrows(table, n); };
        }(i));          // Присвоить значение i локальной переменной n
    }
}
```

15.6.3. Удаление и замена узлов

Метод removeChild() удаляет элемент из дерева документа. Но будьте внимательны: этот метод вызывается не относительно узла, который должен быть удален, а (как следует из фрагмента «child» в его имени) относительно родителя удаляемого узла. Этот метод вызывается относительно родителя и принимает в виде аргумента дочерний узел, который требуется удалить. Чтобы удалить узел n из документа, вызов метода должен осуществляться так:

```
n.parentNode.removeChild(n);
```

Метод replaceChild() удаляет один дочерний узел и замещает его другим. Этот метод должен вызываться относительно родительского узла. В первом аргументе он принимает новый узел, а во втором – замещаемый узел. Например, ниже показано, как заменить узел n текстовой строкой:

```
n.parentNode.replaceChild(document.createTextNode("[ ИСПРАВЛЕНО ]"), n);
```

Следующая функция демонстрирует еще один способ применения метода replaceChild():

```
// Замещает узел n новым элементом <b> и делает узел n его дочерним элементом.
function embolden(n) {
    // Если вместо узла получена строка, интерпретировать ее как значение
    // атрибута id элемента
    if (typeof n == "string") n = document.getElementById(n);
    var parent = n.parentNode;          // Ссылка на родителя элемента n
    var b = document.createElement("b"); // Создать элемент <b>
    parent.replaceChild(b, n);          // Заменить n элементом <b>
    b.appendChild(n);            // Сделать n дочерним элементом элемента <b>
}
```

В разделе 15.5.1 было представлено свойство outerHTML элементов и говорилось, что оно не реализовано в текущей версии Firefox. Пример 15.5 демонстрирует, как можно реализовать это свойство в Firefox (и в любом другом броузере, поддерживающем свойство innerHTML, позволяющем расширять объект-прототип Element.prototype и имеющем методы определения методов доступа к свойствам). Здесь также демонстрируется практическое применение методов removeChild() и cloneNode().

Пример 15.5. Реализация свойства outerHTML с помощью свойства innerHTML

```
// Реализация свойства outerHTML для броузеров, не поддерживающих его.
// Предполагается, что броузер поддерживает свойство innerHTML, возможность
// расширения Element.prototype и позволяет определять методы доступа к свойствам.
(function() {
    // Если свойство outerHTML уже имеется - просто вернуть управление
    if (document.createElement("div").outerHTML) return;

    // Возвращает окружающую разметку с содержимым элемента,
    // на который указывает ссылка this
    function outerHTMLGetter() {
        var container = document.createElement("div");  // Фиктивный элемент
        container.appendChild(this.cloneNode(true));    // Копировать this
                                                        // в фиктивный элемент
        return container.innerHTML;                     // Вернуть содержимое
                                                        // фиктивного элемента
    }

    // Замещает указанным значением value содержимое элемента, на который
    // указывает ссылка this, вместе с окружающей разметкой HTML
    function outerHTMLSetter(value) {
        // Создать фиктивный элемент и сохранить в нем указанное значение
        var container = document.createElement("div");
        container.innerHTML = value;
        // Переместить все узлы из фиктивного элемента в документ
        while(container.firstChild) // Продолжать, пока в контейнере
                                    // не останется дочерних элементов
            this.parentNode.insertBefore(container.firstChild, this);
        // И удалить замещаемый узел
        this.parentNode.removeChild(this);
    }

    // Использовать эти две функции в качестве методов чтения и записи свойства outerHTML
    // всех объектов Element. Использовать метод Object.defineProperty, если имеется,
    // в противном случае использовать __defineGetter__ и __defineSetter__.
    if (Object.defineProperty) {
        Object.defineProperty(Element.prototype, "outerHTML", {
                                get: outerHTMLGetter,
                                set: outerHTMLSetter,
                                enumerable: false, configurable: true
                             });
    }
    else {
        Element.prototype.__defineGetter__("outerHTML", outerHTMLGetter);
        Element.prototype.__defineSetter__("outerHTML", outerHTMLSetter);
    }
}());
```

15.6.4. Использование объектов DocumentFragment

Объекты DocumentFragment – это особая разновидность объектов Node; они служат временным контейнером для других узлов. Создаются объекты DocumentFragment следующим образом:

```
var frag = document.createDocumentFragment();
```

Как и узел Document, объекты DocumentFragment являются самостоятельными и не входят в состав какого-либо другого документа. Его свойство parentNode всегда возвращает значение null. Однако, как и узлы Element, объекты DocumentFragment могут иметь любое количество дочерних элементов, которыми можно управлять с помощью методов appendChild(), insertBefore() и т. д.

Одна из особенностей объекта DocumentFragment состоит в том, что он позволяет манипулировать множеством узлов как единственным узлом: если объект DocumentFragment передать методу appendChild(), insertBefore() или replaceChild(), в документ будут вставлены дочерние элементы фрагмента, а не сам фрагмент. (Дочерние элементы будут перемещены из фрагмента в документ, а сам фрагмент опустеет и будет готов для повторного использования.) Следующая функция использует объект DocumentFragment для перестановки в обратном порядке дочерних элементов узла:

```
// Выполняет перестановку дочерних элементов узла n в обратном порядке
function reverse(n) {
    // Создать пустой объект DocumentFragment, который будет играть роль
    // временного контейнера
    var f = document.createDocumentFragment();
    // Выполнить обход дочерних элементов в обратном порядке и переместить каждый
    // из них в объект фрагмента. Последний дочерний элемент узла n станет первым
    // дочерним элементом фрагмента f, и наоборот. Обратите внимание, что при добавлении
    // дочернего элемента в фрагмент f он автоматически удаляется из узла n.
    while(n.lastChild) f.appendChild(n.lastChild);

    // В заключение переместить сразу все дочерние элементы
    // из фрагмента f обратно в узел n.
    n.appendChild(f);
}
```

В примере 15.6 представлена реализация метода insertAdjacentHTML() (раздел 15.5.1) с применением свойства innerHTML и объекта DocumentFragment. В нем также определяются функции вставки разметки HTML с более логичными именами, чем неочевидное insertAdjacentHTML(). Вложенная вспомогательная функция fragment() является, пожалуй, наиболее интересной частью этого примера: она возвращает объект DocumentFragment, содержащий разобранное представление указанной ей строки с разметкой HTML.

Пример 15.6. Реализация метода insertAdjacentHTML() с использованием свойства innerHTML

```
// Этот модуль определяет метод Element.insertAdjacentHTML для броузеров,
// не поддерживающих его, а также определяет переносимые функции вставки HTML,
// имеющие более логичные имена, чем имя insertAdjacentHTML:
// Insert.before(), Insert.after(), Insert.atStart(), Insert.atEnd()
var Insert = (function() {
    // Если элементы имеют собственный метод insertAdjacentHTML, использовать
    // его в четырех функциях вставки HTML, имеющих более понятные имена.
    if (document.createElement("div").insertAdjacentHTML) {
        return {
            before:  function(e,h) {e.insertAdjacentHTML("beforebegin",h);},
            after:   function(e,h) {e.insertAdjacentHTML("afterend",h);},
            atStart: function(e,h) {e.insertAdjacentHTML("afterbegin",h);},
            atEnd:   function(e,h) {e.insertAdjacentHTML("beforeend",h);}
        };
    }

    // Иначе, в случае отсутствия стандартного метода insertAdjacentHTML,
    // реализовать те же самые четыре функции вставки и затем использовать их
    // в определении метода insertAdjacentHTML.
    // Сначала необходимо определить вспомогательный метод, который принимает
    // строку с разметкой HTML и возвращает объект DocumentFragment,
    // содержащий разобранное представление этой разметки.
    function fragment(html) {
        var elt = document.createElement("div");       // Пустой элемент
        var frag = document.createDocumentFragment(); // Пустой фрагмент
        elt.innerHTML = html;                          // Содержимое элемента

        while(elt.firstChild)                          // Переместить все узлы
            frag.appendChild(elt.firstChild);          // из elt в frag
        return frag;                                   // И вернуть frag
    }

    var Insert = {
        before: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html), elt);
        },
        after: function(elt, html) {
            elt.parentNode.insertBefore(fragment(html),elt.nextSibling);
        },
        atStart: function(elt, html) {
            elt.insertBefore(fragment(html), elt.firstChild);
        },
        atEnd: function(elt, html) { elt.appendChild(fragment(html)); }
    };

    // Реализация метода insertAdjacentHTML на основе функций выше
    Element.prototype.insertAdjacentHTML = function(pos, html) {
        switch(pos.toLowerCase()) {
        case "beforebegin": return Insert.before(this, html);
        case "afterend":    return Insert.after(this, html);
        case "afterbegin":  return Insert.atStart(this, html);
        case "beforeend":   return Insert.atEnd(this, html);
        }
    };
    return Insert;    // Вернуть четыре функции вставки
}());
```

15.7. Пример: создание оглавления

Пример 15.7 показывает, как динамически создавать оглавление документа. Он демонстрирует многие концепции работы с документом, описанные в разделах выше: выбор элементов, навигация по документу, установка атрибутов элементов, установка свойства `innerHTML` и создание новых узлов и вставку их в документ. Пример снабжен большим количеством комментариев, которые призваны облегчить понимание программного кода.

Пример 15.7. Автоматическое создание оглавления документа

```
/**
 * TOC.js: создает оглавление документа.
 *
 * Этот модуль регистрирует анонимную функцию, которая вызывается
 * автоматически по окончании загрузки документа. Эта функция сначала
 * отыскивает элемент документа с атрибутом id="TOC". Если такой элемент
 * отсутствует, функция создает его, помещая в начало документа.
 *
 * Затем функция отыскивает все элементы с <h1> по <h6>, интерпретируя их как
 * заголовки разделов и создает оглавление внутри элемента TOC. Функция добавляет
 * номера разделов в каждый заголовок и обертывает заголовки именованными
 * якорными элементами, благодаря чему оглавление может ссылаться на них.
 * Якорным элементам даются имена, начинающиеся с приставки "TOC", поэтому вам следует
 * избегать использовать эту приставку в своей разметке HTML.
 *
 * Оформление элементов оглавления можно реализовать с помощью CSS. Все элементы имеют
 * класс "TOCEntry". Кроме того, каждый элемент оглавления имеет класс, соответствующий
 * уровню заголовка раздела. Для заголовков, оформленных тегом <h1>, создаются элементы
 * оглавления с классом "TOCLevel1",для заголовков <h2> - с классом "TOCLevel2" и т.д.
 * Номера разделов, вставляемые в заголовки, получают класс "TOCSectNum".
 *
 * Этот модуль можно использовать с каскадными таблицами стилей, такими как:
 *
 * #TOC { border: solid black 1px; margin: 10px; padding: 10px; }
 * .TOCEntry { font-family: sans-serif; }
 * .TOCEntry a { text-decoration: none; }
 * .TOCLevel1 { font-size: 16pt; font-weight: bold; }
 * .TOCLevel2 { font-size: 12pt; margin-left: .5in; }
 * .TOCSectNum:after { content: ": "; }
 *
 * Последнее определение генерирует двоеточие и пробел после номера раздела.
 * Чтобы скрыть номера разделов, можно использовать следующий стиль:
 *
 * .TOCSectNum { display: none }
 *
 * Этот модуль использует вспомогательную функцию onLoad().
 **/
onLoad(function() { // Анонимная функция, определяющая локальн. обл. видимости
    // Отыскать контейнерный элемент для оглавления.
    // Если такой элемент отсутствует, создать его в начале документа.
    var toc = document.getElementById("TOC");
    if (!toc) {
        toc = document.createElement("div");
        toc.id = "TOC";
        document.body.insertBefore(toc, document.body.firstChild);
    }

    // Отыскать все элементы заголовков разделов
    var headings;
    if (document.querySelectorAll) // Возможно есть более простой путь?
        headings = document.querySelectorAll("h1,h2,h3,h4,h5,h6");
    else                        // Иначе отыскать заголовки более сложным способом
        headings = findHeadings(document.body, []);

    // Выполняет рекурсивный обход тела документа в поисках заголовков
    function findHeadings(root, sects) {
        for(var c = root.firstChild; c != null; c = c.nextSibling) {
            if (c.nodeType !== 1) continue;
            if (c.tagName.length == 2 && c.tagName.charAt(0) == "H")
                sects.push(c);
            else
                findHeadings(c, sects);
        }
        return sects;
    }

    // Инициализировать массив, хранящий номера разделов.
    var sectionNumbers = [0,0,0,0,0,0];

    // Выполнить цикл по найденным элементам заголовков.
    for(var h = 0; h < headings.length; h++) {
        var heading = headings[h];

        // Пропустить заголовки, находящиеся в контейнере оглавления.
        if (heading.parentNode == toc) continue;

        // Определить уровень заголовка.
        var level = parseInt(heading.tagName.charAt(1));
        if (isNaN(level) || level < 1 || level > 6) continue;

        // Увеличить номер раздела для этого уровня и установить
        // номера разделов более низкого уровня равными нулю.
        sectionNumbers[level-1]++;
        for(var i = level; i < 6; i++) sectionNumbers[i] = 0;

        // Объединить номера разделов всех уровней,
        // чтобы получился номер вида 2.3.1.
        var sectionNumber = sectionNumbers.slice(0,level).join(".")

        // Добавить номер раздела в заголовок. Номер помещается в элемент <span>,
        // чтобы его можно было стилизовать с помощью CSS.
        var span = document.createElement("span");
        span.className = "TOCSectNum";
        span.innerHTML = sectionNumber;
        heading.insertBefore(span, heading.firstChild);

        // Обернуть заголовок якорным элементом, чтобы можно было
        // сконструировать ссылку на него.
        var anchor = document.createElement("a");
        anchor.name = "TOC"+sectionNumber;
        heading.parentNode.insertBefore(anchor, heading);
        anchor.appendChild(heading);

        // Создать ссылку на этот раздел.
        var link = document.createElement("a");
        link.href = "#TOC" + sectionNumber; // Адрес назначения ссылки
        link.innerHTML = heading.innerHTML; // Текст ссылки совпадает
                                            // с текстом заголовка
        // Поместить ссылку в элемент div, чтобы обеспечить возможность
        // стилизации в зависимости от уровня.
        var entry = document.createElement("div");
        entry.className = "TOCEntry TOCLevel" + level;
        entry.appendChild(link);

        // И добавить элемент div в контейнер оглавления.
        toc.appendChild(entry);
    }
});
```

15.8. Геометрия документа и элементов и прокрутка

До сих пор в этой главе мы рассматривали документы как некие абстрактные деревья элементов и текстовых узлов. Но когда броузер отображает документ в своем окне, он создает визуальное представление документа, в котором каждый элемент имеет определенную позицию и размеры. Часто веб-приложения могут интерпретировать документы как деревья элементов и никак не заботиться о том, как эти элементы отображаются на экране. Однако иногда бывает необходимо определить точные геометрические характеристики элемента. Например, в главе 16 мы увидим, что элемент можно поместить в определенную позицию с помощью CSS. Если вам потребуется использовать CSS для динамического позиционирования элемента (такого как всплывающая подсказка или сноска) рядом с элементом, который позиционируется броузером, вам необходимо иметь возможность определять положение этого элемента.

В этом разделе рассказывается, как можно переходить от абстрактной, древовидной модели документа к геометрическому, основанному на системе координат визуальному представлению документа в окне броузера, и обратно. Свойства и методы, описываемые здесь, реализованы в броузерах достаточно давно (хотя есть некоторые, которые до недавнего времени присутствовали только в IE, а некоторые не были реализованы в IE до появления версии IE9). К моменту написания этих строк они проходили процесс стандартизации консорциумом W3C в виде стандарта «CSSOM-View Module» (*http://www.w3.org/TR/cssom-view/*).

15.8.1. Координаты документа и видимой области

Позиция элемента измеряется в пикселах. Координата X растет слева направо, а координата Y – сверху вниз. Однако существуют две точки, которые мы можем считать началом координат: координаты X и Y элемента могут отсчитываться относительно верхнего левого угла документа или относительно верхнего левого угла видимой области. Для окон верхнего уровня и вкладок «видимой областью» является часть окна броузера, в которой фактически отображается содержимое документа: в нее не входит обрамление окна (меню, панели инструментов, вкладки). Для документов, отображаемых во фреймах, видимой областью является элемент `<iframe>`, определяющий фрейм. В любом случае, когда речь заходит о позиции элемента, необходимо знать, какая система координат используется – относительно начала документа или относительно начала видимой области. (Координаты видимой области видимости иногда называют оконными координатами.)

Если документ меньше видимой области или если он еще не прокручивался, верхний левый угол документа находится в верхнем левом углу видимой области, и начала систем координат документа и видимой области совпадают. Однако в общем случае, чтобы перейти от одной системы координат к другой, необходимо добавлять или вычитать *смещения прокрутки*. Например, если координата Y элемента имеет значение 200 пикселов в системе координат документа и пользователь прокрутил документ вниз на 75, то в системе координат видимой области координата Y элемента будет иметь значение 125 пикселов. Аналогично, если координата X элемента имеет значение 400 в системе координат видимой области и пользователь прокрутил документ по горизонтали на 200 пикселов, то в системе координат документа координата X элемента будет иметь значение 600.

Система координат документа является более фундаментальной, чем система координат видимой области, и на нее не оказывает влияния величина прокрутки. Однако в клиентских сценариях довольно часто используются координаты видимой области. Система координат документа используется при позиционировании элементов с помощью CSS (глава 16). Но проще всего получить координаты элемента в системе координат видимой области (раздел 15.8.2). Аналогично, когда регистрируется функция-обработчик событий от мыши, координаты указателя мыши передаются ей в системе координат видимой области.

Чтобы перейти от одной системы координат к другой, необходимо иметь возможность определять позиции полос прокрутки окна броузера. Во всех броузерах, кроме IE версии 8 и ниже, эти значения можно узнать с помощью свойств `pageXOffset` и `pageYOffset` объекта `Window`. Кроме того, в IE (и во всех современных броузерах) позиции полос прокрутки можно узнать с помощью свойств `scrollLeft` и `scrollTop`. Проблема состоит в том, что в обычном случае позиции полос прокрутки следует читать из свойств корневого элемента (`document.documentElement`) документа, а в режиме совместимости (раздел 13.4.4) необходимо обращаться к элементу `<body>` (`document.body`) документа. Пример 15.8 показывает, как определять позиции полос прокрутки переносимым образом.

Пример 15.8. Получение позиций полос прокрутки окна

```
// Возвращает текущие позиции полос прокрутки в виде свойств x и y объекта
function getScrollOffsets(w) {
    // Использовать указанное окно или текущее,
    // если функция вызвана без аргумента
    w = w || window;

    // Следующий способ работает во всех броузерах, кроме IE версии 8 и ниже
    if (w.pageXOffset != null) return {x: w.pageXOffset, y:w.pageYOffset};

    // Для IE (и других броузеров) в стандартном режиме
    var d = w.document;
    if (document.compatMode == "CSS1Compat")
        return {x:d.documentElement.scrollLeft,
                y:d.documentElement.scrollTop};

    // Для броузеров в режиме совместимости
    return { x: d.body.scrollLeft, y: d.body.scrollTop };
}
```

Иногда бывает удобно иметь возможность определять размеры видимой области, например, чтобы определить, какая часть документа видима в настоящий момент. Как и в случае со смещениями прокрутки, самый простой способ узнать размеры видимой области не работает в IE версии 8 и ниже, а прием, который работает в IE, зависит от режима, в котором отображается документ. Пример 15.9 показывает, как переносимым способом определять размер видимой области. Обратите внимание на сходство программного кода этого примера с программным кодом в примере 15.8.

Пример 15.9. Получение размеров видимой области документа

```
// Возвращает размеры видимой области в виде свойств w и h объекта
function getViewportSize(w) {
    // Использовать указанное окно или текущее, если функция вызвана без аргумента
    w = w || window;

    // Следующий способ работает во всех броузерах, кроме IE версии 8 и ниже
    if (w.innerWidth != null) return {w: w.innerWidth, h:w.innerHeight};

    // Для IE (и других броузеров) в стандартном режиме
    var d = w.document;
    if (document.compatMode == "CSS1Compat")
        return { w: d.documentElement.clientWidth,
                 h: d.documentElement.clientHeight };

    // Для броузеров в режиме совместимости
    return { w: d.body.clientWidth, h: d.body.clientWidth };
}
```

В двух примерах выше использовались свойства `scrollLeft`, `scrollTop`, `clientWidth` и `clientHeight`. Мы встретимся с этими свойствами еще раз в разделе 15.8.5.

15.8.2. Определение геометрии элемента

Самый простой способ определить размеры и координаты элемента – обратиться к его методу `getBoundingClientRect()`. Этот метод впервые появился в IE5 и в настоящее время реализован во всех текущих броузерах. Он не принимает аргументов и возвращает объект со свойствами `left`, `right`, `top` и `bottom`. Свойства `left` и `top` возвращают координаты X и Y верхнего левого угла элемента, а свойства `right` и `bottom` возвращают координаты правого нижнего угла.

Этот метод возвращает позицию элемента в системе координат видимой области. (Слово «Client» в имени метода `getBoundingClientRect()` косвенно указывает на клиентскую область веб-броузера, т. е. на окно и видимую область в нем.) Чтобы перейти к координатам относительно начала документа, которые не изменяются после прокрутки окна броузера пользователем, нужно добавить смещения прокрутки:

```
var box = e.getBoundingClientRect(); // Координаты в видимой области
var offsets = getScrollOffsets();    // Вспомогат. функция, объявленная выше
var x = box.left + offsets.x;        // Перейти к координатам документа
var y = box.top + offsets.y;
```

Кроме того, во многих броузерах (и в стандарте W3C) объект, возвращаемый методом `getBoundingClientRect()`, имеет свойства `width` и `height`, но оригинальная реализация в IE не поддерживает их. Для совместимости ширину и высоту элемента можно вычислять, как показано ниже:

```
var box = e.getBoundingClientRect();
var w = box.width || (box.right - box.left);
var h = box.height || (box.bottom - box.top);
```

В главе 16 вы узнаете, что содержимое элемента окружается необязательной пустой областью, которая называется *отступом* (*padding*). Отступы окружаются необязательной рамкой (border), а рамка окружается необязательными полями (margins). Координаты, возвращаемые методом `getBoundingClientRect()`, включают рамку и отступы элемента, но не включают поля.

Если слово «Client» в имени метода `getBoundingClientRect()` определяет систему координат возвращаемого прямоугольника, то о чем свидетельствует слово «Bounding» (ограничивающий)? Блочные элементы, такие как изображения, абзацы и элементы `<div>`, всегда отображаются броузерами в прямоугольных областях. Однако строчные элементы, такие как `<span>`, `<code>` и `<b>`, могут занимать несколько строк и таким образом состоять из нескольких прямоугольных областей. Например, представьте некоторый курсивный текст (отмеченный тегами `<i>` и `</i>`), разбитый на две строки. Область, занимаемая этим текстом, состоит из прямоугольника в правой части первой строки и прямоугольника в левой части второй строки (в предположении, что текст записывается слева направо). Если передать методу `getBoundingClientRect()` строчный элемент, он вернет геометрию «ограничивающего прямоугольника» (bounding rectangle), содержащего все отдельные прямоугольные области. Для элемента `<i>`, взятого в качестве примера выше, ограничивающий прямоугольник будет включать обе строки целиком.

Для определения координат и размеров отдельных прямоугольников, занимаемых строчными элементами, можно воспользоваться методом `getClientRects()`, который возвращает объект, подобный массиву, доступный только для чтения, чьи элементы представляют объекты прямоугольных областей, подобные тем, что возвращаются методом `getBoundingClientRect()`.

Мы уже знаем, что методы модели DOM, такие как `getElementsByTagName()`, возвращают «живые» результаты, изменяющиеся синхронно с изменением документа. Объекты прямоугольных областей (и списки объектов прямоугольных областей), возвращаемые методами `getBoundingClientRect()` и `getClientRects()` *не являются* «живыми». Они хранят статические сведения о визуальном представлении документа на момент вызова. Они не обновляются, если пользователь прокрутит документ или изменит размеры окна броузера.

15.8.3. Определение элемента в указанной точке

Метод `getBoundingClientRect()` позволяет узнать текущую позицию элемента в видимой области. Но иногда бывает необходимо решить обратную задачу – узнать, какой элемент находится в заданной точке внутри видимой области. Сделать это можно с помощью метода `elementFromPoint()` объекта `Document`. Он принимает координаты X и Y (относительно начала координат видимой области, а не документа) и возвращает объект `Element`, находящийся в этой позиции. На момент написания этих строк алгоритм выбора элемента не был строго определен, но суть реализации метода сводится к тому, что он должен возвращать самый внутренний и самый верхний (в смысле CSS-атрибута `z-index`, который описывается в разделе 16.2.1.1) элемент, находящийся в этой точке. Если передать ему координаты точки, находящейся за пределами видимой области, метод `elementFromPoint()` вернет `null`, даже если после преобразования координат в систему координат документа получится вполне допустимая точка.

Метод `elementFromPoint()` выглядит весьма практичным, и наиболее очевидный случай его использования – определение элемента, находящегося под указателем мыши по его координатам. Однако, как будет показано в главе 17, объект события от мыши уже содержит эту информацию в своем свойстве `target`. Именно поэтому на практике метод `elementFromPoint()` почти не используется.

15.8.4. Прокрутка

В примере 15.8 демонстрировалось, как определять позиции полос прокрутки окна броузера. Чтобы заставить броузер прокрутить документ, можно присваивать значение используемым в этом примере свойствам `scrollLeft` и `scrollTop`, но существует более простой путь, поддерживаемый с самых ранних дней развития языка JavaScript. Метод `scrollTo()` объекта `Window` (и его синоним `scroll()`) принимает координаты X и Y точки (относительно начала координат документа) и устанавливает их в качестве величин смещения полос прокрутки. То есть он прокручивает окно так, что точка с указанными координатами оказывается в верхнем левом углу видимой области. Если указать точку, расположенную слишком близко к нижней или к правой границе документа, броузер попытается поместить эту точку как можно ближе к верхнему левому углу видимой области, но не сможет обеспечить точное их совпадение. Следующий пример прокручивает окно броузера так, что видимой оказывается самая нижняя часть документа:

```
// Получить высоту документа и видимой области. Метод offsetHeight описывается ниже.
var documentHeight = document.documentElement.offsetHeight;
var viewportHeight = window.innerHeight; // Или использовать getViewportSize(),
                                         // описанный выше
// И прокрутить окно так, чтобы переместить последнюю "страницу" в видимую область
window.scrollTo(0, documentHeight - viewportHeight);
```

Метод `scrollBy()` объекта `Window` похож на методы `scroll()` и `scrollTo()`, но их аргументы определяют относительное смещение и добавляются к текущим позициям полос прокрутки. Тем, кто умеет быстро читать, мог бы понравиться следующий букмарклет (раздел 13.2.5.1):

```
// Прокручивает документ на 10 пикселов каждые 200 мсек.
// Учтите, что этот букмарклет нельзя отключить после запуска!
javascript:void setInterval(function() {scrollBy(0,10)}, 200);
```

Часто требуется прокрутить документ не до определенных числовых координат, а до элемента в документе, который нужно сделать его видимым. В этом случае можно определить координаты элемента с помощью метода `getBoundingClientRect()`, преобразовать их в координаты относительно начала документа и передать их методу `scrollTo()`, но гораздо проще воспользоваться методом `scrollIntoView()` требуемого HTML-элемента. Этот метод гарантирует, что элемент, относительно которого он будет вызван, окажется в видимой области. По умолчанию он старается прокрутить документ так, чтобы верхняя граница элемента оказалась как можно ближе к верхней границе видимой области. Если в единственном аргументе передать методу значение `false`, он попытается прокрутить документ так, чтобы нижняя граница элемента совпала с нижней границей видимой области. Кроме того, броузер выполнит прокрутку по горизонтали, если это потребуется, чтобы сделать элемент видимым.

Своим поведением метод `scrollIntoView()` напоминает свойство `window.location.hash`, когда ему присваивается имя якорного элемента (элемента `<a name="">`).

15.8.5. Подробнее о размерах, позициях и переполнении элементов

Метод `getBoundingClientRect()` поддерживается всеми текущими броузерами, но если требуется обеспечить поддержку броузеров более старых версий, этот метод использовать нельзя и для определения размеров и позиций элементов следует применять более старые приемы. Размеры элементов определяются достаточно просто: доступные только для чтения свойства `offsetWidth` и `offsetHeight` любого HTML-элемента возвращают его размеры в пикселах. Возвращаемые размеры включают рамку элемента и отступы, но не включают поля, окружающие рамку снаружи.

Все HTML-элементы имеют свойства `offsetLeft` и `offsetTop`, возвращающие их координаты X и Y. Для многих элементов эти координаты откладываются относительно начала документа и непосредственно определяют позицию элемента. Но для потомков позиционируемых элементов и некоторых других элементов, таких как ячейки таблиц, эти свойства возвращают координаты относительно элемента-предка, а не документа. Свойство `offsetParent` определяет, относительно какого элемента исчисляются значения этих свойств. Если `offsetParent` имеет значение `null`, свойства содержат координаты относительно начала документа. Таким образом, в общем случае для определения позиции элемента `e` с помощью его свойств `offsetLeft` и `offsetTop` требуется выполнить цикл:

```
function getElementPosition(e) {
    var x = 0, y = 0;
    while(e != null) {
        x += e.offsetLeft;
        y += e.offsetTop;
        e = e.offsetParent;
    }
    return {x:x, y:y};
}
```

Выполняя обход цепочки, образуемой ссылками `offsetParent`, и накапливая смещения, эта функция вычисляет координаты указанного элемента относительно начала документа. (Напомню, что метод `getBoundingClientRect()`, напротив, возвращает координаты относительно начала видимой области.) Однако на этом тему позиционирования элементов нельзя считать исчерпанной – функция `getElementPosition()` не всегда вычисляет правильные значения, и ниже будет показано, как исправить эту ошибку.

В дополнение к множеству свойств `offset` все элементы документа определяют еще две группы свойств; имена свойств в первой группе начинаются с приставки `client`, а во второй группе – с приставки `scroll`. То есть каждый HTML-элемент имеет все свойства, перечисленные ниже:

```
offsetWidth   clientWidth   scrollWidth
offsetHeight  clientHeight  scrollHeight
offsetLeft    clientLeft    scrollLeft
offsetTop     clientTop     scrollTop
offsetParent
```

Чтобы понять разницу между группами свойств `client` и `scroll`, необходимо уяснить, что объем содержимого HTML-элемента может не умещаться в прямоугольную область, выделенную для этого содержимого, и поэтому отдельные элементы могут иметь собственные полосы прокрутки (смотрите описание CSS-атрибута `overflow` в разделе 16.2.6). Область отображения содержимого – это видимая область, подобная видимой области в окне броузера, и когда содержимое элемента не умещается в видимой области, необходимо принимать в учет позиции полос прокрутки элемента.

Свойства `clientWidth` и `clientHeight` похожи на свойства `offsetWidth` и `offsetHeight`, за исключением того, что они включают только область содержимого и отступы и не включают размер рамки. Кроме того, если броузер добавляет между рамкой и отступами полосы прокрутки, то свойства `clientWidth` и `clientHeight` не включают ширину полос прокрутки в возвращаемые значения. Обратите внимание, что для строчных элементов, таких как `<i>`, `<code>` и `<span>`, свойства `clientWidth` и `clientHeight` всегда возвращают 0.

Свойства `clientWidth` и `clientHeight` использовались в методе `getViewportSize()` из примера 15.9. В том случае, когда эти свойства применяются к корневому элементу документа (или телу элемента в режиме совместимости), они возвращают те же значения, что и свойства `innerWidth` и `innerHeight` окна.

Свойства `clientLeft` и `clientTop` не имеют большой практической ценности: они возвращают расстояние по горизонтали и вертикали между внешней границей отступов элемента и внешней границей его рамки. Обычно эти значения просто определяют ширину левой и верхней рамки. Однако если элемент имеет полосы прокрутки и если броузер помещает эти полосы прокрутки вдоль левого или верхнего края (что весьма необычно), значения свойств `clientLeft` и `clientTop` также будут включать ширину полос прокрутки. Для строчных элементов свойства `clientLeft` и `clientTop` всегда возвращают 0.

Свойства `scrollWidth` и `scrollHeight` определяют размер области содержимого элемента, плюс его отступы, плюс ширину и высоту области содержимого, выходящую за видимую область. Когда содержимое целиком умещается в видимой области, значения этих свойств совпадают со значениями свойств `clientWidth` и `clientHeight`. В противном случае они включают ширину и высоту области содержимого, выходящую за видимую область, и возвращают значения, превосходящие значения свойств `clientWidth` и `clientHeight`.

Наконец, свойства `scrollLeft` и `scrollTop` определяют позиции полос прокрутки элемента. Мы использовали эти свойства корневого элемента документа в методе `getScrollOffsets()` (пример 15.8), но они также присутствуют в любом другом элементе. Обратите внимание, что свойства `scrollLeft` и `scrollTop` доступны для записи и им можно присваивать значения, чтобы прокрутить содержимое элемента. (HTML-элементы не имеют метода `scrollTo()`, как объект `Window`.)

Когда документ содержит прокручиваемые элементы, содержимое которых не умещается в видимой области, объявленный выше метод `getElementPosition()` дает некорректные результаты из-за того, что он не учитывает позиции полос прокрутки. Ниже приводится исправленная версия, которая вычитает позиции полос прокрутки из накопленных смещений, т. е. преобразует координаты относительно начала документа в координаты относительно видимой области:

```
function getElementPos(elt) {
    var x = 0, y = 0;
    // Цикл, накапливающий смещения
    for(var e = elt; e != null; e = e.offsetParent) {
        x += e.offsetLeft;
        y += e.offsetTop;
    }
    // Еще раз обойти все элементы-предки, чтобы вычесть смещения прокрутки.
    // Он также вычтет позиции главных полос прокрутки и преобразует
    // значения в координаты видимой области.
    for(var e=elt.parentNode; e != null && e.nodeType == 1; e=e.parentNode) {
        x -= e.scrollLeft;
        y -= e.scrollTop;
    }
    return {x:x, y:y};
}
```

В современных броузерах этот метод `getElementPos()` возвращает те же координаты, что и метод `getBoundingClientRect()` (но он менее эффективен). Теоретически такая функция, как `getElementPos()`, могла бы использоваться в броузерах, не поддерживающих метод `getBoundingClientRect()`. Однако броузеры, не поддерживающие `getBoundingClientRect()`, имеют массу несовместимостей в механизме позиционирования элементов, и поэтому в них такая простая функция оказывается ненадежной. Клиентские библиотеки, такие как jQuery, включают функции вычисления позиций элементов, которые дополняют этот простой алгоритм множеством проверок, учитывающих несовместимости между броузерами. Если вам потребуется определять позиции элементов в броузерах, не поддерживающих `getBoundingClientRect()`, то для этих целей лучше использовать библиотеку, например, jQuery.

15.9. HTML-формы

HTML-элемент `<form>` и различные элементы ввода, такие как `<input>`, `<select>` и `<button>`, занимают видное место в разработке клиентских сценариев. Эти HTML-элементы появились в самом начале развития Всемирной паутины, еще до появления языка JavaScript. Формы HTML – это механизм веб-приложений первого поколения, не требующий применения JavaScript. Ввод пользователя собирается в элементах форм; затем форма отправляется на сервер; сервер обрабатывает ввод и генерирует новую HTML-страницу (обычно с новыми элементами форм) для отображения на стороне клиента.

Элементы HTML-форм по-прежнему остаются великолепным инструментом получения данных от пользователя, даже когда данные формы целиком обрабатываются JavaScript-сценарием на стороне клиента и не должны отправляться на сервер. С точки зрения программиста, разрабатывающего серверные сценарии, форма оказывается совершенно бесполезной, если в ней отсутствует кнопка отправки формы. Однако с точки зрения программиста, разрабатывающего клиентские сценарии, кнопка отправки вообще не нужна (хотя все еще может использоваться). Серверные программы опираются на механизм отправки форм – они обрабатывают данные порциями, объем которых определяется формой, – и это ограничивает их интерактивные возможности. Клиентские программы опираются на механизм событий – они могут откликаться на события, возникающие в отдельных элементах форм, – и это позволяет обеспечить более высокую степень интерактивности. Например, клиентская программа может проверять ввод пользователя по мере нажатия клавиш. Или откликаться на выбор флажка, разрешая доступ к набору параметров, которые имеют смысл, только когда флажок отмечен.

В следующих подразделах описывается, как реализовать подобные операции при работе с HTML-формами. Формы конструируются из HTML-элементов, как и любые другие части HTML-документа, и ими можно управлять с помощью методов модели DOM, описанных выше в этой главе. Элементы формы стали первыми элементами, для которых было предусмотрено программное управление на самых ранних этапах развития клиентского JavaScript, поэтому они поддерживают некоторые дополнительные функции, предшествовавшие появлению DOM.

Обратите внимание, что в этом разделе описываются не сам язык разметки HTML, а приемы управления HTML-формами. Здесь предполагается, что вы уже имеете некоторое знакомство с HTML-элементами (`<input>`, `<textarea>`, `<select>` и т. д), используемыми для создания форм. Тем не менее в табл. 15.1 для справки приводится список наиболее часто используемых элементов форм. Дополнительные сведения о функциях для работы с формами и элементами форм можно узнать в четвертой части книги, в справочных статьях `Form`, `Input`, `Option`, `Select` и `TextArea`.

Таблица 15.1. Элементы HTML-форм

HTML-элемент	Свойство type	Обработчик событий	Описание и события

`<input type="button">`
или
`<button type="button">`
	`"button"`	`onclick`	Кнопка

`<input type="checkbox">`	`"checkbox"`	`onchange`	Переключаемый флажок; с иным поведением, чем у радиокнопки
`<input type="file">`	`"file"`	`onchange`	Поле ввода имени файла для выгрузки на сервер; свойство `value` доступно только для чтения
`<input type="hidden">`	`"hidden"`	нет	Данные, отправляемые вместе с формой, но не видимые пользователю
`<option>`	нет	нет	Единственный пункт объекта `Select`; обработчик событий подключается к объекту `Select`, а не к объектам `Option`
`<input type="password">`	`"password"`	`onchange`	Поле ввода пароля – скрывает вводимые символы
`<input type="radio">`	`"radio"`	`onchange`	Переключатель с поведением радиокнопки – в каждый конкретный момент времени может быть выбран только один переключатель

`<input type="reset">`
или
`<button type="reset">`
	`"reset"`	`onclick`	Кнопка, сбрасывающая форму в исходное состояние

`<select>`	`"select-one"`	`onchange`	Список или раскрывающееся меню, в котором можно выбрать только один пункт (смотрите также `<option>`)
`<select multiple>`	`"select-multiple"`	`onchange`	Список, в котором можно выбрать сразу несколько пунктов (смотрите также `<option>`)

Таблица 15.1 (продолжение)

HTML-элемент	Свойство type	Обработчик событий	Описание и события

`<input type="submit">`
или
`<button type="submit">`
	"submit"	onclick	Кнопка, инициирующая отправку формы

`<input type="text">`	"text"	onchange	Однострочное текстовое поле ввода; по умолчанию в элементе `<input>` атрибут type отсутствует или не опознается
`<textarea>`	"textarea"	onchange	Многострочное текстовое поле ввода

15.9.1. Выбор форм и элементов форм

Формы и элементы, содержащиеся в них, можно выбрать с помощью стандартных методов, таких как `getElementById()` и `getElementsByTagName()`:

```
var fields = document.getElementById("address").getElementsByTagName("input");
```

В броузерах, поддерживающих `querySelectorAll()`, можно выбрать все радиокнопки или все элементы с одинаковыми именами, присутствующие в форме, как показано ниже:

```
// Все радиокнопки в форме с атрибутом id="shipping"
document.querySelectorAll('#shipping input[type="radio"]');
// Все радиокнопки с атрибутом name="method" в форме с атрибутом id="shipping"
document.querySelectorAll('#shipping input[type="radio"][name="method"]');
```

Однако, как описывалось в разделах 14.7, 15.2.2 и 15.2.3, элемент `<form>` с установленным атрибутом `name` или `id` можно также выбрать другими способами. Элемент `<form>` с атрибутом `name="address"` можно выбрать любым из следующих способов:

```
window.address         // Ненадежный: старайтесь не использовать
document.address       // Может применяться только к формам с атрибутом name
document.forms.address // Явное обращение к форме с атрибутом name или id
document.forms[n]      // Ненадежный: n - порядковый номер формы
```

В разделе 15.2.3 говорилось, что свойство `document.forms` ссылается на объект `HTMLCollection`, позволяющий выбирать элементы `<form>` по их порядковым номерам, по значению атрибута `id` или `name`. Объекты `Form` сами по себе действуют подобно объектам `HTMLCollection`, хранящим элементы форм, и могут индексироваться именами или числами. Если первый элемент формы с атрибутом `name="address"` имеет атрибут `name="street"`, на него можно сослаться с помощью любого из следующих выражений:

```
document.forms.address[0]
document.forms.address.street
document.address.street // только для name="address", но не id="address"
```

Если необходимо явно указать, что выполняется обращение к элементу формы, вместо объекта формы можно индексировать его свойство `elements`:

```
document.forms.address.elements[0]
document.forms.address.elements.street
```

Для выбора конкретных элементов документа предпочтительнее использовать атрибут `id`. Однако при отправке форм атрибут `name` играет особую роль и чаще используется с самими формами, чем с элементами форм. Обычно при работе с группами флажков и обязательно при работе с группами радиокнопок для атрибута `name` используется одно и то же значение. Напомню, что, когда объект `HTMLCollection` индексируется именем и существует сразу несколько элементов, использующих одно и то же имя, возвращаемым значением является объект, подобный массиву, содержащий все элементы с указанным именем. Взгляните на следующую форму, содержащую радиокнопки для выбора метода доставки товара:

```
<form name="shipping">
  <fieldset><legend>Shipping Method</legend>
    <label><input type="radio" name="method" value="1st">Первым классом</label>
    <label><input type="radio" name="method" value="2day">
      За 2 дня самолетом
    </label>
    <label><input type="radio" name="method" value="overnite">
      В течение ночи
    </label>
  </fieldset>
</form>
```

Сослаться на массив радиокнопок в этой форме можно следующим образом:

```
var methods = document.forms.shipping.elements.method;
```

Обратите внимание, что элементы `<form>` имеют HTML-атрибут и соответствующее ему свойство с именем «method», поэтому в данном случае необходимо использовать свойство `elements` формы вместо прямого обращения к свойству `method`. Чтобы определить, какой метод доставки выбрал пользователь, необходимо обойти элементы формы в массиве и проверить свойство `checked` каждого из них:

```
var shipping_method;
for(var i = 0; i < methods.length; i++)
    if (methods[i].checked) shipping_method = methods[i].value;
```

Со свойствами элементов форм, такими как `checked` и `value`, мы поближе познакомимся в следующем разделе.

15.9.2. Свойства форм и их элементов

Наиболее интересным для нас свойством объекта `Form` является массив `elements[]`, описанный выше. Остальные свойства объекта `Form` менее важны. Свойства `action`, `encoding`, `method` и `target` непосредственно соответствуют атрибутам `action`, `encoding`, `method` и `target` элемента `<form>`. Все эти свойства и атрибуты используются для управления отправкой данных формы на веб-сервер и отображением результатов. Клиентский сценарий на языке JavaScript может устанавливать значения этих свойств, но это имеет смысл, только когда форма действительно отправляется серверной программе.

До появления JavaScript отправка форм выполнялась с помощью специальной кнопки `Submit`, а сброс значений элементов формы в значения по умолчанию –

с помощью специальной кнопки `Reset`. В языке JavaScript тем же целям служат два метода, `submit()` и `reset()`, объекта `Form`. Метод `submit()` объекта `Form` отправляет форму, а метод `reset()` сбрасывает элементы формы в исходное состояние.

У всех (или у большинства) элементов форм есть общие свойства, перечисленные далее. Кроме того, у некоторых элементов есть специальные свойства, которые будут описаны ниже, когда мы будем рассматривать элементы форм различных типов по отдельности.

`type`

Доступная только для чтения строка, идентифицирующая тип элемента формы. Для элементов форм, определяемых с помощью тега `<input>`, это свойство просто хранит значение атрибута `type`. Другие элементы форм (такие как `<textarea>` и `<select>`) также определяют свойство `type`, благодаря чему его можно использовать в сценарии для идентификации элементов, подобно тому, как идентифицируются различные типы элементов `<input>`. Значения этого свойства для каждого типа элементов форм перечислены во втором столбце табл. 15.1.

`form`

Доступная только для чтения ссылка на объект `Form`, в котором содержится этот элемент, или `null`, если элемент не находится внутри элемента `<form>`.

`name`

Доступная только для чтения строка, указанная в HTML-атрибуте `name`.

`value`

Доступная для чтения и записи строка, определяющая «значение», содержащееся в элементе формы или представляемое им. Эта строка отсылается на веб-сервер при передаче формы и только иногда представляет интерес для JavaScript-программ. Для элементов `Text` и `Textarea` это свойство содержит введенный пользователем текст. Для кнопок, создаваемых с помощью тега `<input>` (но не для кнопок, создаваемых с помощью тега `<button>`), это свойство определяет отображаемый на кнопке текст. Свойство `value` для элементов переключателей (радиокнопок) и флажков не редактируется и никак не представляется пользователю. Это просто строка, устанавливаемая HTML-атрибутом `value`. Эта строка предназначена для отправки веб-серверу, и ее можно использовать для передачи дополнительных данных. Свойство `value` будет обсуждаться далее в этой главе, когда мы будем рассматривать различные категории элементов формы.

15.9.3. Обработчики событий форм и их элементов

Каждый элемент `Form` имеет обработчик события `onsubmit`, возникающего в момент отправки формы, и обработчик события `onreset`, возникающего в момент сброса формы в исходное состояние. Обработчик `onsubmit` вызывается непосредственно перед отправкой формы. Он может отменить отправку, вернув значение `false`. Это дает JavaScript-программам возможность проверить ввод пользователя и избежать отправки неполных или ошибочных данных серверной программе. Обратите внимание, что обработчик `onsubmit` вызывается только в случае щелчка мышью на кнопке `Submit`. Вызов метода `submit()` формы не приводит к вызову обработчика `onsubmit`.

Обработчик событий `onreset` похож на обработчик `onsubmit`. Он вызывается непосредственно перед сбросом формы в исходное состояние и может предотвратить сброс элементов формы, вернув значение `false`. Кнопки `Reset` редко используются в формах, но если у вас имеется такая кнопка, возможно, у вас появится желание запросить у пользователя подтверждение, прежде чем выполнить сброс:

```
<form...
    onreset="return confirm('Вы действительно хотите сбросить все и начать сначала?')">
    ...
    <button type="reset">Очистить поля ввода и начать сначала</button>
</form>
```

Подобно обработчику `onsubmit`, обработчик `onreset` вызывается только в случае щелчка мышью на кнопке `Reset`. Вызов метода `reset()` формы не приводит к вызову обработчика `onreset`.

Элементы форм обычно возбуждают событие `click` или `change`, когда пользователь взаимодействует с ними, и вы можете реализовать обработку этих событий, определив обработчик `onclick` или `onchange`. В третьем столбце таблицы 15.1 для каждого элемента формы указан основной обработчик событий. Вообще говоря, элементы форм, являющиеся кнопками, возбуждают событие `click` в момент активации (даже когда активация производится посредством нажатия клавиши на клавиатуре, а не щелчком мышью). Другие элементы форм возбуждают событие `change`, когда пользователь изменяет содержимое, представляемое элементом. Это происходит, когда пользователь вводит текст в текстовое поле или выбирает элемент раскрывающегося списка. Обратите внимание, что это событие возбуждается не каждый раз, когда пользователь нажимает клавишу, находясь в текстовом поле ввода. Оно возбуждается, только когда пользователь изменит значение элемента и перенесет фокус ввода в другой элемент. То есть этот обработчик событий вызывается по завершении ввода. Радиокнопки и флажки являются кнопками, хранящими информацию о своем состоянии, и все они возбуждают события `click` и `change`; из них событие `change` имеет большее практическое значение.

Элементы форм также возбуждают событие `focus`, когда они получают фокус ввода, и событие `blur`, когда теряют его.

Важно знать, что внутри обработчика события ключевое слово `this` всегда ссылается на элемент документа, вызвавший данное событие (подробнее об этом будет рассказываться в главе 17). Во всех элементах форм имеется свойство `form`, ссылающееся на форму, в которой содержится элемент, поэтому обработчики событий элемента формы всегда могут обратиться к объекту `Form`, как к `this.form`. Сделав еще один шаг, мы можем сказать, что обработчик событий для одной формы может ссылаться на соседний элемент формы, имеющий имя `x`, как `this.form.x`.

15.9.4. Кнопки

Кнопки являются одними из наиболее часто используемых элементов форм, т. к. они предоставляют понятный визуальный способ вызова пользователем какого-либо запрограммированного сценарием действия. Элемент кнопки не имеет собственного поведения, предлагаемого по умолчанию, и не представляет никакой пользы без обработчика события `onclick`. Кнопки, определяемые с помощью элементов `<input>`, отображают простой текст, содержащийся в их свойстве `value`.

Кнопки, определяемые с помощью элементов `<button>`, отображают содержимое элемента.

Обратите внимание, что гиперссылки предоставляют такой же обработчик событий `onclick`, как и кнопки. Когда действие, выполняемое обработчиком `onclick`, можно классифицировать как «переход по ссылке», используйте ссылку. В противном случае используйте кнопку.

Элементы `Submit` и `Reset` очень похожи на элементы-кнопки, но имеют связанные с ними действия, предлагаемые по умолчанию (передача или очистка формы). Если обработчик событий `onclick` возвращает `false`, стандартное действие этих кнопок, предусмотренное по умолчанию, не выполняется. Обработчик `onclick` элемента `Submit` можно использовать для проверки введенных значений, но обычно это делается в обработчике `onsubmit` самой формы.

В четвертой части книги нет описания элемента `Button`. Описание всех элементов-кнопок, включая описание элементов, создаваемых с помощью тега `<button>`, вы найдете в разделе, посвященном элементу `Input`.

15.9.5. Переключатели и флажки

Флажки и радиокнопки имеют два визуально различимых состояния: они могут быть либо установлены, либо сброшены. Пользователь может изменить состояние такого элемента, щелкнув на нем. Радиокнопки обычно объединяются в группы связанных элементов, имеющих одинаковые значения HTML-атрибута `name`. При установке одной из радиокнопок предыдущая установленная в группе радиокнопка сбрасывается. Флажки тоже часто объединяются в группы с общим значением атрибута `name`, и, когда вы обращаетесь к ним по имени, необходимо помнить, что вы получаете в ответ объект, подобный массиву, а не отдельный элемент.

И флажки, и переключатели имеют свойство `checked`. Это доступное для чтения и записи логическое значение определяет, отмечен ли элемент в данный момент. Свойство `defaultChecked` представляет собой доступное только для чтения логическое значение, содержащее значение HTML-атрибута `checked`; оно определяет, должен ли элемент отмечаться, когда страница загружается в первый раз.

Флажки и радиокнопки сами не отображают какой-либо текст и обычно выводятся вместе с прилегающим к ним HTML-текстом (или со связанным тегом `<label>`). Это значит, что установка свойства `value` элемента флажка или радиокнопки не изменяет внешнего вида элемента. Свойство `value` можно установить, но это изменит лишь строку, отсылаемую на веб-сервер при передаче данных формы.

Когда пользователь щелкает на флажке или радиокнопке, элемент вызывает свой обработчик `onclick`. Если состояние флажка или радиокнопки изменяется в результате щелчка мышью, они также вызывают обработчик событий `onchange`. (При этом радиокнопки, изменяющие состояние в результате щелчка на другой радиокнопке, не вызывают обработчик `onchange`.)

15.9.6. Текстовые поля ввода

Однострочные текстовые поля ввода `Text` применяются в HTML-формах и JavaScript-программах, пожалуй, чаще других. Они позволяют пользователю ввести короткий однострочный текст. Свойство `value` представляет текст, введенный пользователем. Установив это свойство, можно явно задать выводимый текст.

Определяемый стандартом HTML5 атрибут placeholder позволяет указать строку приглашения к вводу, которая будет отображаться в поле ввода до того момента, пока пользователь не введет какой-нибудь текст:

```
Дата прибытия: <input type="text" name="arrival" placeholder="yyyy-mm-dd">
```

Обработчик событий onchange текстового поля ввода вызывается, когда пользователь вводит новый текст или редактирует существующий, а затем указывает, что он завершил редактирование, убрав фокус ввода из текстового поля.

Элемент Textarea (многострочное текстовое поле ввода) очень похож на элемент Text за исключением того, что разрешает пользователю ввести (а JavaScript-программе вывести) многострочный текст. Многострочное текстовое поле создается тегом <textarea>, синтаксис которого существенно отличается от синтаксиса тега <input>, используемого для создания однострочного текстового поля. (Подробнее об этом см. в разделе с описанием элемента Textarea в четвертой части книги.) Тем не менее эти два типа элементов ведут себя очень похожим образом. Свойство value и обработчик событий onchange элемента Textarea можно использовать точно так же, как в случае с элементом Text.

Элемент <input type="password"> – это модификация однострочного текстового поля ввода, в котором вместо вводимого пользователем текста отображаются символы звездочек. Как можно заключить из имени элемента, его можно использовать, чтобы дать пользователю возможность вводить пароли, не беспокоясь о том, что другие прочитают их через плечо. Следует понимать, что элемент Password защищает введенные пользователем данные от любопытных глаз, но при отправке данных формы эти данные никак не шифруются (если только отправка не выполняется по безопасному HTTPS-соединению) и при передаче по сети могут быть перехвачены.

Наконец, элемент <input type="file"> предназначен для ввода пользователем имени файла, который должен быть выгружен на веб-сервер. По существу, это однострочное текстовое поле, совмещенное со встроенной кнопкой, выводящей диалог выбора файла. У элемента выбора файла, как и у однострочного текстового поля, есть обработчик событий onchange. Однако, в отличие от текстового поля ввода, свойство value элемента выбора файла доступно только для чтения. Это не дает злонамеренным JavaScript-программам обмануть пользователя, выгрузив файл, не предназначенный для отправки на сервер.

Различные текстовые элементы ввода определяют обработчики событий onkeypress, onkeydown и onkeyup. Можно вернуть false из обработчиков событий onkeypress или onkeydown, чтобы запретить обработку нажатой пользователем клавиши. Это может быть полезно, например, когда требуется заставить пользователя вводить только цифры. Этот прием демонстрируется в примере 17.6.

15.9.7. Элементы Select и Option

Элемент Select представляет собой набор вариантов (представленных элементами Option), которые могут быть выбраны пользователем. Броузеры обычно отображают элементы Select в виде раскрывающихся меню, но, если указать в атрибуте size значение больше чем 1, они будут отображать их в виде списков (возможно, с полосами прокрутки). Элемент Select может работать двумя сильно различающимися способами, а выбор того или иного способа определяется значением свойства type.

Если в теге <select> определен атрибут `multiple`, пользователь сможет выбрать несколько вариантов, а свойство `type` объекта `Select` будет иметь значение «select-multiple». В противном случае, если атрибут `multiple` отсутствует, может быть выбран только один вариант и свойство `type` будет иметь значение «select-one».

В некотором смысле элемент с возможностью множественного выбора похож на набор флажков, а элемент без такой возможности – на набор радиокнопок. Однако варианты выбора, отображаемые элементом `Select`, не являются кнопками-переключателями: они определяются с помощью тега <option>. Элемент `Select` определяет свойство `options`, которое является объектом, подобным массиву, хранящим объекты `Option`.

Когда пользователь выбирает тот или иной вариант или отменяет выбор, элемент `Select` вызывает свой обработчик событий `onchange`. Для элементов `Select` с возможностью выбора единственного варианта, доступное только для чтения свойство `selectedIndex` определяет выбранный в данный момент вариант. Для элементов `Select` с возможностью множественного выбора одного свойства `selectedIndex` недостаточно для представления полного набора выбранных вариантов. В этом случае для определения выбранных вариантов следует в цикле перебрать элементы массива `options[]` и проверить значения свойства `selected` каждого объекта `Option`.

Кроме свойства `selected` у каждого объекта `Option` есть свойство `text`, задающее строку текста, которая отображается в элементе `Select` для данного варианта. Используя это свойство, можно изменить видимый пользователем текст. Свойство `value` представляет доступную для чтения и записи строку текста, который отсылается на веб-сервер при передаче данных формы. Даже если вы пишете исключительно клиентскую программу и ваши формы никуда не отправляются, свойство `value` (или соответствующий ей HTML-атрибут `value`) можно использовать для хранения данных, которые потребуются после выбора пользователем определенного варианта. Обратите внимание, что элемент `Option` не определяет связанных с формой обработчиков событий; используйте вместо этого обработчик `onchange` соответствующего элемента `Select`.

Помимо задания свойства `text` объектов `Option` имеются способы динамического изменения выводимых в элементе `Select` вариантов с помощью особых возможностей свойства `options`, которые ведут свое существование с самого начала появления поддержки клиентских сценариев. Можно обрезать массив элементов `Option`, установив свойство `options.length` равным требуемому количеству вариантов, или удалить все объекты `Option`, установив значение свойства `options.length` равным нулю. Можно удалять отдельные объекты `Option` из элемента `Select`, присваивая элементам массива `options[]` значение `null`. В этом случае удаляются соответствующие объекты `Option`, а все элементы, расположенные в массиве `options[]` правее, автоматически сдвигаются влево, заполняя опустевшее место.

Чтобы добавить в элемент `Select` новый вариант, можно создать его с помощью конструктора `Option()` и добавить в конец массива `options[]`, как показано ниже:

```
// Создать новый объект Option
var zaire = new Option("Zaire",  // Свойство text
                       "zaire",  // Свойство value
                       false,    // Свойство defaultSelected
                       false);   // Свойство selected

// Отобразить вариант в элементе Select, добавив его в конец массива options:
```

```
var countries = document.address.country; // Получить объект Select
countries.options[countries.options.length] = zaire;
```

Имейте в виду, что эти специальные возможности элемента Select пришли к нам из прошлых времен. Вставку и удаление вариантов можно реализовать более очевидным способом, воспользовавшись стандартными методами `Document.createElement()`, `Node.insertBefore()`, `Node.removeChild()` и другими.

15.10. Другие особенности документов

Эта глава начиналась с утверждения, что она является одной из наиболее важных в книге. Она также по необходимости оказалась одной из наиболее длинных. Этот раздел завершает главу описанием некоторых особенностей объекта Document.

15.10.1. Свойства объекта Document

В этой главе уже были представлены некоторые свойства объекта Document, такие как `body`, `documentElement` и `forms`, ссылающиеся на специальные элементы документа. Кроме них документы определяют еще несколько свойств, представляющих интерес:

`cookie`

Специальное свойство, позволяющее JavaScript-программам читать и писать cookie-файлы. Это свойство рассматривается в главе 20.

`domain`

Свойство, которое позволяет доверяющим друг другу веб-серверам, принадлежащим одному домену, ослаблять ограничения, связанные с политикой общего происхождения, на взаимодействие между их веб-страницами (подробности см. в разделе 13.6.2.1).

`lastModified`

Строка, содержащая дату последнего изменения документа.

`location`

Это свойство ссылается на тот же объект Location, что и свойство `location` объекта Window.

`referrer`

URL-адрес документа, содержащего ссылку (если таковая существует), которая привела броузер к текущему документу. Это свойство имеет то же значение, что и HTTP-заголовок Referer, но записывается с двумя буквами r.

`title`

Текст между тегами `<title>` и `</title>` данного документа.

`URL`

Свойство URL документа является строкой, доступной только для чтения, а не объектом Location. Значение этого свойства совпадает с начальным значением свойства `location.href`, но, в отличие от объекта Location, не является динамическим. Если пользователь выполнит переход, указав новый идентификатор фрагмента внутри документа, то свойство `location.href` изменится, а свойство `document.URL` – нет.

Из всех этих свойств наибольший интерес представляет свойство `referrer`: оно содержит URL-адрес документа, содержащего ссылку, которая привела пользователя к текущему документу. Это свойство можно было бы использовать, как показано ниже:

```
if (document.referrer.indexOf("http://www.google.com/search?") == 0) {
    var args = document.referrer.substring(ref.indexOf("?")+1).split("&");
    for(var i = 0; i < args.length; i++) {
        if (args[i].substring(0,2) == "q=") {
            document.write("<p>Добро пожаловать, пользователь Google. ");
            document.write("Вы искали: " +
                           unescape(args[i].substring(2)).replace('+', ' ');
            break;
        }
    }
}
```

Метод `document.write()`, использованный в этом примере, является темой следующего раздела.

15.10.2. Метод document.write()

Метод `document.write()` был одним из первых методов, реализованных еще в веб-броузере Netscape 2. Он появился еще до создания модели DOM и представлял единственный способ отображения динамически изменяемого текста в документе. В современных сценариях надобность в этом методе отпала, но вы наверняка встретите его в существующем программном коде.

Метод `document.write()` объединяет свои строковые аргументы и вставляет получившуюся строку в документ, в точку вызова метода. По завершении выполнения сценария броузер выполнит синтаксический анализ сгенерированного вывода и отобразит его. Например, следующий фрагмент использует метод `write()` для динамического вывода информации в статический HTML-документ:

```
<script>
    document.write("<p>Заголовок документа: " + document.title);
    document.write("<br>URL: " + document.URL);
    document.write("<br>Ссылающийся на него документ: " + document.referrer);
    document.write("<br>Изменен: " + document.lastModified);
    document.write("<br>Открыт: " + new Date());
</script>
```

Важно понимать, что метод `write()` можно использовать для вывода разметки HTML только в процессе синтаксического анализа документа. То есть вызывать метод `document.write()` из программного кода верхнего уровня в теге `<script>` можно только в случае, если выполнение сценария является частью процесса анализа документа. Если поместить вызов `document.write()` в определение функции и затем вызвать эту функцию из обработчика события, результат окажется неожиданным – этот вызов уничтожит текущий документ и все содержащиеся в нем сценарии! (Почему это происходит, будет описано чуть ниже.) По тем же причинам нельзя использовать `document.write()` в сценариях с установленными атрибутами `defer` или `async`.

Пример 13.3 в главе 13 использует метод `document.write()` указанным способом, чтобы сгенерировать более сложный вывод.

Метод `write()` можно использовать для создания полностью новых документов в других окнах и фреймах. (Однако при работе с другими окнами и фреймами необходимо позаботиться о том, чтобы не нарушать политику общего происхождения.) Первый вызов метода `write()` другого документа полностью уничтожит имеющееся содержимое этого документа. Метод `write()` можно вызывать многократно, тем самым добавляя новое содержимое в документ. Содержимое, передаваемое методу `write()`, может буферизоваться (и не отображаться), пока не будет выполнен завершающий вызов метода `close()` объекта документа, сообщающий парсеру, что работа с документом окончена и он может выполнить разбор документа и отобразить его.

Следует отметить, что объект `Document` поддерживает также метод `writeln()`, который идентичен методу `write()`, за исключением того, что он добавляет символ перевода строки после вывода своих аргументов. Это может пригодиться, например, при выводе форматированного текста внутри элемента `<pre>`.

Метод `document.write()` редко используется в современных сценариях: свойство `innerHTML` и другие приемы, поддерживаемые моделью DOM, обеспечивают более удачные способы добавления содержимого в документ. С другой стороны, некоторые алгоритмы лучше укладываются в схему потокового ввода/вывода, реализуемую методом `write()`. Если вы создаете сценарий, который динамически генерирует и выводит текст в процессе своего выполнения, вам, возможно, будет интересно ознакомиться с примером 15.10, в котором свойство `innerHTML` указанного элемента обертывается простыми методами `write()` и `close()`.

Пример 15.10. Интерфейс потоков ввода-вывода к свойству innerHTML

```
// Определить простейший интерфейс "потоков ввода/вывода" для свойства innerHTML элемента.
function ElementStream(elt) {
    if (typeof elt === "string") elt = document.getElementById(elt);
    this.elt = elt;
    this.buffer = "";
}

// Объединяет все аргументы и добавляет в буфер
ElementStream.prototype.write = function() {
    this.buffer += Array.prototype.join.call(arguments, "");
};

// То же, что и write(), но добавляет символ перевода строки
ElementStream.prototype.writeln = function() {
    this.buffer += Array.prototype.join.call(arguments, "") + "\n";
};

// Переносит содержимое буфера в элемент и очищает буфер.
ElementStream.prototype.close = function() {
    this.elt.innerHTML = this.buffer;
    this.buffer = "";
};
```

15.10.3. Получение выделенного текста

Иногда удобно иметь возможность определять, какой участок текста документа выделен пользователем. Сделать это можно, как показано ниже:

```
function getSelectedText() {
    if (window.getSelection)     // Функция, определяемая стандартом HTML5
        return window.getSelection().toString();
    else if (document.selection) // Прием, характерный для IE.
        return document.selection.createRange().text;
}
```

Стандартный метод `window.getSelection()` возвращает объект `Selection`, описывающий текущий выделенный текст, как последовательность одного или более объектов `Range`. Объекты `Selection` и `Range` определяют чрезвычайно сложный прикладной интерфейс, который практически не используется и не описывается в этой книге. Наиболее важной и широко реализованной (везде, кроме IE) особенностью объекта `Selection` является его метод `toString()`, который возвращает простое текстовое содержимое выделенной области.

Броузер IE определяет иной прикладной интерфейс, который не описывается в этой книге. Метод `document.selection` возвращает объект, представляющий выделенную область. Метод `createRange()` этого объекта возвращает реализованный только в IE объект `TextRange`, свойство `text` которого содержит выделенный текст.

Прием, подобный приведенному в примере выше, в частности, может пригодиться в букмарклетах (раздел 13.2.5.1) для организации поиска выделенного текста в поисковых системах или на сайте. Так, следующая HTML-ссылка пытается отыскать текущий выделенный фрагмент текста в Википедии. Если поместить в закладку эту ссылку и URL-адрес со спецификатором `javascript:`, закладка превратится в букмарклет:

```
<a href="javascript: var q;
    if (window.getSelection) q = window.getSelection().toString();
    else if (document.selection) q = document.selection.createRange().text;
    void window.open('http://ru.wikipedia.org/wiki/' + q);">
  Поиск выделенного текста в Википедии
</a>
```

В примере выше, выбирающем выделенный текст, есть одна проблема, связанная с несовместимостью. Метод `getSelection()` объекта `Window` не возвращает выделенный текст, если он находится внутри элемента `<input>` или `<textarea>`: он возвращает только тот текст, который выделен в теле самого документа. В то же время свойство `document.selection`, поддерживаемое броузером IE, возвращает текст, выделенный в любом месте в документе.

Чтобы получить текст, выделенный в текстовом поле ввода или в элементе `<textarea>`, можно использовать следующее решение:

```
elt.value.substring(elt.selectionStart, elt.selectionEnd);
```

Свойства `selectionStart` и `selectionEnd` не поддерживаются в версиях IE8 и ниже.

15.10.4. Редактируемое содержимое

Мы уже познакомились с элементами форм, включая текстовые поля ввода и элементы textarea, которые дают пользователям возможность вводить и редактировать простой текст. Вслед за IE все текущие веб-броузеры стали поддерживать простые средства редактирования разметки HTML. Вы можете увидеть их в действии на страницах (например, на страницах блогов, где можно оставлять комментарии), встраивающих редактор форматированного текста с панелью инструментов, содержащей кнопки для выбора стиля отображения шрифта (полужирный, курсив), настройки выравнивания и вставки изображений и ссылок.

Существует два способа включения поддержки возможности редактирования. Можно установить HTML-атрибут contenteditable любого тега или установить JavaScript-свойство contenteditable соответствующего элемента Element, содержимое которого разрешается редактировать. Когда пользователь щелкнет на содержимом внутри этого элемента, появится текстовый курсор и пользователь сможет вводить текст с клавиатуры. Ниже представлен HTML-элемент, создающий область редактирования:

```
<div id="editor" contenteditable>
    Щелкните здесь, чтобы отредактировать
</div>
```

Броузеры могут поддерживать автоматическую проверку орфографии для полей форм и элементов с атрибутом contenteditable. В броузерах, поддерживающих такую проверку, она может быть включена или выключена по умолчанию. Чтобы явно включить ее, следует добавить атрибут spellcheck. А чтобы явно запретить – добавить атрибут spellcheck=false (если, например, в элементе <textarea> предполагается выводить программный код или другой текст с идентификаторами, отсутствующими в словаре).

Точно так же можно сделать редактируемым весь документ, записав в свойство designMode объекта Document строку «on». (Чтобы снова сделать документ доступным только для чтения, достаточно записать в это свойство строку «off».) Свойство designMode не имеет соответствующего ему HTML-атрибута. Можно сделать документ доступным для редактирования, поместив его в элемент <iframe>, как показано ниже (обратите внимание, что здесь используется функция onLoad() из примера 13.5):

```
<iframe id="editor" src="about:blank"></iframe>     // Пустой фрейм
<script>
onLoad(function() {                                 // После загрузки
    var editor = document.getElementById("editor"); // найти фрейм документа
    editor.contentDocument.designMode = "on";       // и включить режим
});                                                 // редактирования.
</script>
```

Все текущие броузеры поддерживают свойства contenteditable и designMode. Однако они оказываются плохо совместимыми, когда дело доходит до фактического редактирования. Все броузеры позволяют вставлять и удалять текст и перемещать текстовый курсор с помощью клавиатуры и мыши. Во всех броузерах нажатие клавиши Enter выполняет переход на новую строку, но разные броузеры создают в результате разную разметку. Некоторые начинают новый абзац, другие просто вставляют элемент `<br/>`.

Некоторые броузеры позволяют использовать горячие комбинации клавиш, такие как Ctrl-B, чтобы изменить шрифт выделенного текста на полужирный. В других броузерах (таких как Firefox) стандартные для текстовых процессоров комбинации, такие как Ctrl-B и Ctrl-I, выполняют другие операции, имеющие отношение к самому броузеру, а не к текстовому редактору.

Броузеры определяют множество команд редактирования текста, для большинства из которых не предусмотрены горячие комбинации клавиш. Чтобы выполнить эти команды, необходимо использовать метод `execCommand()` объекта `Document`. (Обратите внимание, что это метод объекта `Document`, а не элемента с атрибутом `contenteditable`. Если документ содержит более одного редактируемого элемента, команда применяется к тому из них, в котором в текущий момент находится текстовый курсор.) Команды, выполняемые методом `execCommand()`, определяются строками, такими как «bold», «subscript», «justifycenter» или «insertimage». Имя команды передается методу `execCommand()` в первом аргументе. Некоторые команды требуют дополнительное значение. Например, команда «createlink» требует указать URL для гиперссылки. Теоретически, если во втором аргументе передать методу `execCommand()` значение `true`, броузер автоматически запросит у пользователя ввести необходимое значение. Однако для большей совместимости вам необходимо самим запрашивать у пользователя требуемые данные, передавая `false` во втором аргументе и требуемое значение – в третьем аргументе.

Ниже приводятся две функции, которые реализуют редактирование с помощью метода `execCommand()`:

```
function bold() { document.execCommand("bold", false, null); }
function link() {
    var url = prompt("Введите адрес гиперссылки");
    if (url) document.execCommand("createlink", false, url);
}
```

Команды, выполняемые методом `execCommand()`, обычно запускаются кнопками на панели инструментов. Качественный пользовательский интерфейс должен запрещать доступ к кнопкам, если вызываемые ими команды недоступны. Чтобы определить, поддерживается ли некоторая команда броузером, можно передать ее имя методу `document.queryCommandSupported()`. Вызовом метода `document.queryCommandEnabled()` можно узнать, доступна ли команда в настоящее время. (Команда, которая выполняет некоторые действия с выделенным текстом, например, может быть недоступна, пока не будет выделен фрагмент текста.) Некоторые команды, такие как «bold» и «italic», могут иметь логическое состояние «включено» или «выключено» в зависимости от наличия выделенного фрагмента текста или местоположения текстового курсора. Как правило, эти команды представлены на панели инструментов кнопками-переключателями. Для определения текущего состояния таких команд можно использовать метод `document.queryCommandState()`. Наконец, некоторые команды, такие как «fontname», ассоциируются с некоторым значением (именем семейства шрифтов). Узнать это значение можно с помощью метода `document.queryCommandValue()`. Если в текущем выделенном фрагменте используются шрифты двух разных семейств, команда «fontname» будет иметь неопределенное значение. Для проверки этого случая можно использовать метод `document.queryCommandIndeterm()`.

Различные броузеры реализуют различные наборы команд редактирования. Некоторые команды, такие как «bold», «italic», «createlink», «undo» и «redo», поддерживаются всеми броузерами.[1] На момент написания этих строк проект стандарта HTML5 определял команды, перечисленные ниже. Однако, поскольку они реализованы пока не во всех броузерах, здесь не будет даваться сколько-нибудь подробное их описание:

```
bold           insertLineBreak       selectAll
createLink     insertOrderedList     subscript
delete         insertUnorderedList   superscript
formatBlock    insertParagraph       undo
forwardDelete  insertText            unlink
insertImage    italic                unselect
insertHTML     redo
```

Если вашему веб-приложению потребуется обеспечить возможность редактирования форматированного текста, то вам, вероятно, лучше обратить внимание на уже реализованные решения, учитывающие различия между броузерами. В Интернете можно найти множество готовых компонентов редакторов.[2] Следует отметить, что функция редактирования, встроенная в броузеры, обладает достаточно широкими возможностями, чтобы дать пользователю возможность вводить небольшие объемы форматированного текста, но она оказывается недостаточно мощной для создания серьезных документов. Обратите внимание, в частности, что разметка HTML, генерируемая этими редакторами, весьма далека от идеала.

После того как пользователь отредактирует содержимое элемента с атрибутом `contenteditable`, можно воспользоваться свойством `innerHTML`, чтобы получить разметку HTML отредактированного содержимого. Что дальше делать с полученным отформатированным текстом, полностью зависит от вас. Его можно сохранить в скрытом поле формы и отправить вместе с формой на сервер. Непосредственную отправку отредактированного текста на сервер можно выполнить с помощью приемов, описываемых в главе 18. Можно также сохранить результаты редактирования локально, с помощью механизмов, описываемых в главе 20.

---

[1] Список поддерживаемых команд можно найти по адресу *http://www.quirksmode.org/dom/execCommand.html*.

[2] Фреймворки YUI и Dojo включают такие компоненты редакторов. Список других решений можно найти на странице *http://en.wikipedia.org/wiki/Online_rich-text_editor*.

16

Каскадные таблицы стилей

Каскадные таблицы стилей (Cascading Style Sheets, CSS) – это стандарт визуального представления HTML-документов. Каскадные таблицы стилей предназначены для использования дизайнерами: они позволяют точно определить шрифты, цвета, величину полей, выравнивание, параметры рамок и даже координаты элементов в документе. Но они также представляют интерес и для программистов, пишущих на клиентском языке JavaScript, потому что позволяют воспроизводить анимационные эффекты, такие как плавное появление содержимого документа из-за правого края, например, или сворачивание и разворачивание списков, благодаря чему пользователь получает возможность управлять объемом отображаемой информации. Когда подобные визуальные эффекты только появились, они казались революционными. Совместное применение технологий CSS и JavaScript, обеспечивающее получение разнообразных визуальных эффектов, получило не совсем удачное название «динамический язык HTML» (Dynamic HTML, DHTML), которое уже вышло из употребления.

CSS – достаточно сложный стандарт, который на момент написания этих строк продолжал активно развиваться. Тема CSS настолько объемная, что для полного ее охвата потребовалось бы написать отдельную книгу, и ее детальное обсуждение выходит далеко за рамки *этой* книги.[1] Однако чтобы освоить принципы работы с каскадными таблицами стилей, совершенно необходимо знать хотя бы основы CSS и наиболее часто используемые стили. Поэтому эта глава начинается с обзора CSS, за которым следует описание ключевых стилей, наиболее часто используемых в сценариях. Вслед за этими двумя вводными разделами глава переходит к описанию принципов работы с CSS. В разделе 16.3 описываются наиболее общие и важные приемы: изменение стилей, применяемых к отдельным элементам документа с помощью HTML-атрибута `style`. Атрибут `style` элемента можно использовать для задания стилей, но с его помощью нельзя узнать текущие настройки стилей элементов. В разделе 16.4 рассказывается, как можно получить *вычисленные стили* любого элемента. В разделе 16.5 вы узнаете, как можно изменить сразу несколько стилей путем изменения атрибута `style` элемента. Имеется также возможность непосредственно управлять таблицами стилей, хотя это и редко используется на практике. В разделе 16.6 будет показано, как включать и отключать таблицы стилей, изменять правила в существующих таблицах и добавлять новые таблицы.

---

[1] Рекомендуемое издание: Эрик Мейер «CSS – каскадные таблицы стилей. Подробное руководство», 3-е издание – Пер. с англ. – СПб.: Символ-Плюс, 2008.

16.1. Обзор CSS

Визуальное представление HTML-документов определяется множеством параметров: шрифты, цвета, отступы и т. д. Все эти параметры перечислены в стандарте CSS, где они называются *свойствами стиля* (style properties). Стандарт CSS определяет свойства, которые задают шрифты, цвета, величину полей, параметры рамок, фоновые изображения, выравнивание текста, размеры элементов и их позиции. Чтобы задать визуальное представление HTML-элементов, мы определяем соответствующие значения CSS-свойств. Для этого необходимо записать имя свойства и его значение через двоеточие:

```
font-weight: bold
```

Чтобы полностью описать визуальное представление элемента, обычно бывает необходимо указать значения нескольких свойств. Когда требуется записать несколько пар имя/значение, они отделяются друг от друга точками с запятой:

```
margin-left: 10%;  /* левое поле 10% от ширины страницы */
text-indent: .5in; /* отступ 1/2 дюйма */
font-size: 12pt;   /* размер шрифта 12 пунктов */
```

Как видите, в каскадные таблицы стилей можно помещать комментарии, заключая их в пары символов /* и */. Однако здесь не поддерживаются комментарии, начинающиеся с символов //.

Существует два способа связать набор значений CSS-свойств с HTML-элементами, чье визуальное представление они определяют. Первый заключается в установке атрибута `style` отдельного HTML-элемента. Такие стили называются встроенными:

```
<p style="margin: 20px; border: solid red 2px;">
Этот абзац имеет увеличенные поля и окружен прямоугольной рамкой красного цвета.
</p>
```

Однако намного эффективнее отделять стили CSS от отдельных HTML-элементов и определять их в *таблицах стилей*. Таблица стилей связывает наборы свойств стилей с группами HTML-элементов, которые описываются с помощью *селекторов*. Селектор определяет, или «выбирает», один или более элементов документа, опираясь на атрибут `id` или `class`, имя тега или на другие более специализированные критерии. Селекторы были представлены в разделе 15.2.5, где также было показано, как с помощью `querySelectorAll()` выбрать группу элементов, соответствующих селектору.

Основным элементом таблицы стилей CSS является правило стиля, состоящее из селектора и набора свойств стиля с их значениями, заключенного в фигурные скобки. Таблица стилей может содержать любое количество правил:

```
p {                    /* селектору "p" соответствуют все элементы <p> */
   text-indent: .5in; /* первая строка с отступом 0,5 дюйма */
}
.warning {                    /* Любой элемент с атрибутом class="warning" */
   background-color: yellow; /* будет иметь желтый фон */
   border: solid black 5px;  /* и толстую черную рамку */
}
```

Таблицу стилей CSS можно добавить в HTML-документ, заключив ее в теги `<style>` и `</style>` внутри тега `<head>`. Подобно элементам `<script>`, синтаксический анализ содержимого элементов `<style>` выполняется отдельно и не интерпретируется как разметка HTML:

```
<html>
<head><title>Тестовый документ</title>
<style>
  body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
  p { font-size: 24px; }
</style>
</head>
<body><p>Проверка, проверка</p>
</html>
```

Если таблица стилей используется более чем в одной странице веб-сайта, ее обычно лучше сохранить в отдельном файле, без использования объемлющих HTML-тегов. Затем этот файл CSS можно будет подключить к HTML-странице. В отличие от элемента `<script>`, элемент `<style>` не имеет атрибута `src`. Чтобы подключить таблицу стилей к HTML-странице, следует использовать элемент `<link>` в теге `<head>` документа:

```
<head>
   <title>Тестовый документ</title>
   <link rel="stylesheet" href="mystyles.css" type="text/css">
</head>
```

Мы вкратце рассмотрели, как работают каскадные таблицы стилей. При этом с таблицами CSS связано несколько важных моментов, которые нужно знать и которые описываются в следующих подразделах.

16.1.1. Каскад правил

Напомню, что буква «С» в аббревиатуре CSS обозначает «cascade» (каскадная). Этот термин указывает, что правила стилей, применяемые к конкретному элементу документа, могут быть получены из «каскада» различных источников:

- Таблицы стилей по умолчанию веб-броузера
- Таблицы стилей документа
- Атрибуты `style` отдельных HTML-элементов

Стили, определяемые в атрибуте `style`, переопределяют стили из таблицы стилей, а стили из таблицы стилей переопределяют таблицу стилей броузера, применяемую по умолчанию. Визуальное представление любого элемента может определяться комбинацией свойств стилей из всех трех источников. Более того, элемент может соответствовать сразу нескольким селекторам в таблице стилей. В этом случае к элементу применяются все правила стилей, ассоциированные с такими селекторами. (Если различные селекторы определяют различные значения для одних и тех же свойств стилей, значение, ассоциированное с более конкретным селектором, переопределяет значение, ассоциированное с менее конкретным селектором, однако более подробное обсуждение выходит за рамки этой книги.)

Для отображения любого элемента документа веб-броузер должен объединить атрибут `style` элемента со стилями из всех совпавших селекторов таблицах стилей документа. Результатом этого объединения является фактический набор свойств стилей и значений, которые используются для отображения элемента. Этот набор значений называется *вычисленным стилем* элемента.

16.1.2. История развития CSS

CSS – это довольно старый стандарт. В декабре 1996 года был принят стандарт CSS1 и определены атрибуты для задания цвета, шрифта, полей, рамок и других базовых стилей. Такие старые броузеры, как Netscape 4 и Internet Explorer 4, в значительной степени поддерживают CSS1. Вторая редакция стандарта, CSS2, была принята в мае 1998 года; она определяет более развитые возможности, наиболее важной из которых является возможность абсолютного позиционирования элементов. Версия CSS2.1 дополняет и уточняет положения стандарта CSS2. Из нее были исключены возможности, которые не были реализованы в броузерах. Текущие броузеры практически полностью реализуют поддержку стандарта CSS2.1, хотя в версиях IE, предшествовавших IE8, имеются существенные пробелы.

Работа над CSS продолжается и в настоящее время. Третья версия CSS разделена на специализированные модули, работа по стандартизации которых ведется по отдельности. Спецификации и рабочие проекты CSS можно найти по адресу *http://www.w3.org/Style/CSS/current-work*.

16.1.3. Сокращенная форма определения свойств

Свойства стиля, которые часто используются совместно, допускается объединять вместе, используя сокращенную форму записи. Например, свойства `font-family`, `font-size`, `font-style` и `font-weight` можно определить в виде единственного свойства `font` с составным значением:

```
font: bold italic 24pt helvetica;
```

Аналогично свойства `border`, `margin` и `padding` являются сокращенными именами для свойств, определяющих параметры рамок, полей и отступов (пространство между рамкой и содержимым элемента) для каждой из сторон элемента. Например, чтобы определить параметры рамки для каждой из сторон в отдельности, вместо свойства `border` можно использовать свойства `border-left`, `border-right`, `border-top` и `border-bottom`. Каждое из этих свойств также является сокращенной формой записи. Вместо свойства `border-top` можно определить свойства `border-top-color`, `border-top-style` и `border-top-width`.

16.1.4. Нестандартные свойства

Когда производители броузеров реализуют нестандартные свойства CSS, они добавляют префикс к именам свойств. В броузере Firefox используется префикс `moz-`, в Chrome – `-webkit-`, а в IE – `-ms-`. Производители броузеров добавляют префиксы, даже когда реализуют свойства, которые в будущем, как предполагается, будут включены в стандарт. В качестве примера можно назвать свойство `border-radius`, которое определяет закругленные углы. Это свойство впервые было реализовано как экспериментальное в Firefox 3 и Safari 4, и к его имени были добавлены соответствующие префиксы. Когда стандарт устоялся в достаточной степени, в версиях Firefox 4 и Safari 5 префикс был убран, и теперь они поддерживают свойство `border-radius` без префикса. (Chrome и Opera уже давно поддерживают это свойства без префикса. IE9 также поддерживает его без префикса, но IE8 не поддерживает его, даже с префиксом.)

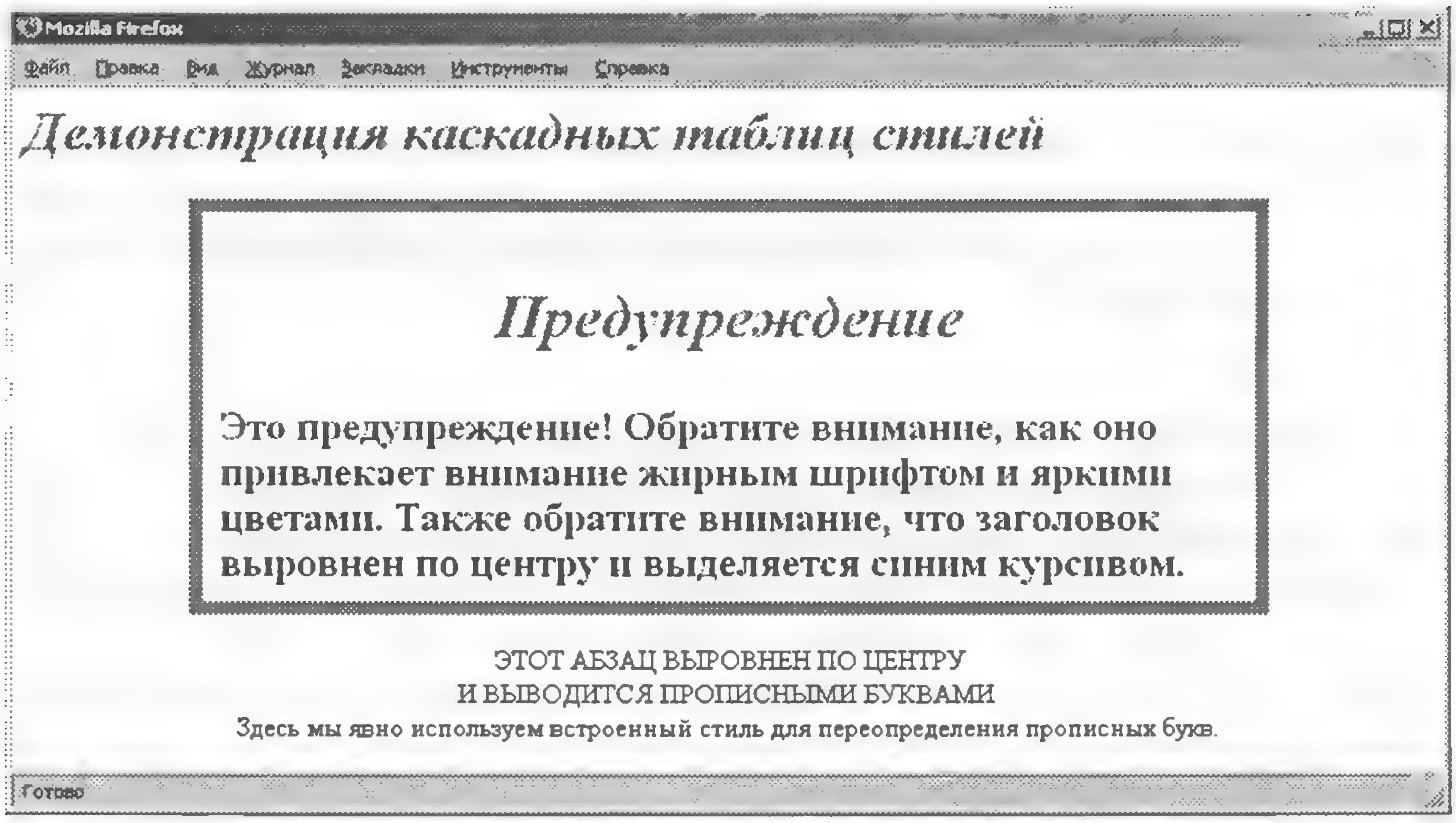
При работе со свойствами CSS, которые в разных броузерах имеют отличающиеся имена, можно определять для таких свойств классы:

```
.radius10 {
  border-radius: 10px;         /* для текущих броузеров */
  -moz-border-radius: 10px;    /* для Firefox 3.x */
  -webkit-border-radius: 10px; /* для Safari 3.2 и 4 */
}
```

Определив такой класс, можно просто добавить значение «radius10» в атрибут `class` любого элемента, чтобы дать элементу рамку с закругленными углами.

16.1.5. Пример CSS-таблицы

Пример 16.1 представляет собой HTML-файл, определяющий и использующий таблицу стилей. Он иллюстрирует селекторы, базирующиеся на имени тега, атрибутах `class` и `id`, а также содержит встроенный стиль, определяемый атрибутом `style`. На рис. 16.1 показано, как этот пример отображается в броузере.



Демонстрация каскадных таблиц стилей

Предупреждение

Это предупреждение! Обратите внимание, как оно привлекает внимание жирным шрифтом и яркими цветами. Также обратите внимание, что заголовок выровнен по центру и выделяется синим курсивом.

ЭТОТ АБЗАЦ ВЫРОВНЕН ПО ЦЕНТРУ
И ВЫВОДИТСЯ ПРОПИСНЫМИ БУКВАМИ
Здесь мы явно используем встроенный стиль для переопределения прописных букв.

Рис. 16.1. *Веб-страница, оформленная с помощью CSS*

Пример 16.1. Определение и использование каскадных таблиц стилей

```
<head>
<style type="text/css">
/* Указывает, что заголовки отображаются курсивом синего цвета. */
h1, h2 { color: blue; font-style: italic }
/*
 * Любой элемент с атрибутом class="WARNING" отображается крупными жирными
 * символами, имеет большие поля и желтый фон с жирной красной рамкой.
 */
.WARNING {
        font-weight: bold;
        font-size: 150%;
        margin: 0 1in 0 1in; /* сверху справа, снизу слева */
        background-color: yellow;
        border: solid red 8px;
        padding: 10px; /* 10 пикселов со всех 4 сторон */
}

/*
 * Текст заголовков h1 и h2 внутри элементов с атрибутом class="WARNING"
 * должен быть выровнен по центру, в дополнение к выделению синим курсивом.
 */
.WARNING h1, .WARNING h2 { text-align: center }
/*
 * Отдельный элемент с атрибутом id="special" отображается
 * прописными буквами по центру.
 */
#special {
        text-align: center;
        text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Демонстрация использования каскадных таблиц стилей</h1>

<div class="WARNING">
<h2>Предупреждение</h2>
Это предупреждение!
Обратите внимание, как оно привлекает внимание жирным шрифтом
и яркими цветами. Также обратите внимание, что заголовок выровнен
по центру и выделяется синим курсивом.
</div>

<p id="special">
Этот абзац выровнен по центру<br>
и выводится прописными буквами.<br>
<span style="text-transform: none">
Здесь мы явно используем встроенный стиль для переопределения прописных букв.
</span>
</p>
```

Ультрасовременные свойства CSS

Когда я работал над этой главой, CSS находился в процессе революционных изменений: производители броузеров реализовали поддержку новых мощных свойств стиля, таких как `border-radius`, `text-shadow`, `box-shadow` и `column-count`. Другой качественно новой возможностью CSS стали *веб-шрифты*: новое CSS-правило `@font-face` позволяет загружать и использовать нестандартные шрифты. (Подробнее о шрифтах, которые свободно могут использоваться в веб-страницах, и о простом механизме их загрузки с серверов компании Google можно прочитать на странице *http://code.google.com/webfonts*.)

Еще одной революционной разработкой в области каскадных таблиц стилей стал модуль «CSS Transitions». Этот проект стандарта определяет возможности, позволяющие преобразовать в анимационные эффекты любые динамические изменения стилей CSS в документе. (Когда поддержка этого стандарта будет реализована достаточно широко, это позволит избавиться от программного кода, воспроизводящего анимационные эффекты, связанные со сменой стилей CSS, как показано в разделе 16.3.1.) Положения модуля «CSS Transitions» реализованы во всех текущих броузерах, кроме IE, но в именах свойств стиля присутствуют префиксы производителей. Проект родственного стандарта «CSS Animations», использующий модуль «CSS Transitions» в качестве основы, определяет более сложные анимационные последовательности. В настоящее время «CSS Animations» реализован только в веб-броузерах, основанных на механизме Webkit. Ни один из этих стандартов не описывается в данной главе, но вам, как веб-разработчикам, нужно знать о существовании этих технологий.

Другим проектом, касающимся CSS, и о котором должны знать веб-разработчики, является стандарт «CSS Transforms», позволяющий определять двухмерные преобразования (вращение, масштабирование, перемещение, а также их комбинации, определяемые в матричном виде), применяемые к любым элементам. Все текущие броузеры (включая версии IE9 и выше) поддерживают этот проект с добавлением приставок, соответствующих производителям. Более того, в Safari реализована поддержка расширения, позволяющего выполнять трехмерные преобразования, но пока неясно, последуют ли этому другие броузеры.

16.2. Наиболее важные CSS-свойства

Для разработчиков клиентских сценариев на языке JavaScript наиболее важными являются CSS-свойства, которые позволяют задавать режим видимости, размер и точную позицию отдельных элементов документа. Другие CSS-свойства дают возможность определять порядок наложения слоев, степень прозрачности, вырезанные области, поля, отступы, рамки и цвета. При работе с CSS-свойствами важно понимать, как работают свойства стиля. Они перечислены в табл. 16.1 и более подробно описываются в последующих разделах.

Таблица 16.1. Наиболее важные CSS-свойства

Свойство	**Описание**
`position`	Определяет тип позиционирования, применяемый к элементу
`top, left`	Позиция верхнего и левого краев элемента
`bottom, right`	Позиция нижнего и правого краев элемента
`width, height`	Размер элемента
`z-index`	«Порядок в стеке» относительно любых перерывающих его элементов (третье измерение в позиционировании элемента)
`display`	Режим отображения элемента
`visibility`	Режим видимости элемента
`clip`	«Область отсечения» элемента (отображаются только те части документа, которые находятся внутри этой области)
`overflow`	Определяет, что следует делать, если размер элемента больше, чем предоставленное ему место
`margin, border, padding`	Границы и рамки элемента
`background`	Цвет фона или фоновый рисунок для элемента
`opacity`	Степень непрозрачности (или прозрачности) элемента. Это свойство относится к стандарту CSS3 и поддерживается не всеми броузерами. Работающая альтернатива имеется для IE

16.2.1. Позиционирование элементов с помощью CSS

CSS-свойство `position` задает тип позиционирования, применяемый к элементу. Это свойство может иметь четыре возможных значения:

`static`

Это значение, применяемое по умолчанию. Оно указывает, что элемент позиционируется статически в соответствии с нормальным порядком вывода содержимого документа (для большинства западных языков – слева направо и сверху вниз). Статически позиционированные элементы не могут позиционироваться с помощью свойств `top`, `left` и других. Для позиционирования элемента документа с применением приемов CSS сначала нужно установить его свойство `position` равным одному из трех других значений.

`absolute`

Это значение позволяет задать абсолютную позицию элемента относительно содержащего его элемента. Такие элементы позиционируются независимо от всех остальных элементов и не являются частью потока статически позиционированных элементов. Абсолютно позиционированный элемент позиционируется либо относительно тела документа, либо, если он вложен в другой абсолютно позиционированный элемент, относительно этого элемента.

`fixed`

Это значение позволяет зафиксировать положение элемента относительно окна броузера. Элементы с фиксированным позиционированием не прокручиваются с остальной частью документа. Как и абсолютно позиционированные, фиксировано позиционированные элементы не зависят от всех остальных элементов и не являются частью потока вывода документа. Фиксированное позиционирование поддерживается большинством современных броузеров, исключая IE6.

`relative`

Если свойство `position` имеет значение `relative`, элемент располагается в соответствии с нормальным потоком вывода, а затем его положение смещается относительно его обычного положения в потоке. Пространство, выделенное для элемента в нормальном потоке вывода документа, остается выделенным для него, и элементы, расположенные со всех сторон от него, не смыкаются для заполнения этого пространства и не «выталкиваются» с новой позиции элемента.

Присвоив свойству `position` элемента значение, отличное от `static`, можно задать положение элемента с помощью произвольной комбинации свойств `left`, `top`, `right` и `bottom`. Чаще всего для позиционирования используются свойства `left` и `top`, задающие расстояние от левого края элемента-контейнера (обычно самого документа) до левого края позиционируемого элемента и расстояние от верхнего края контейнера до верхнего края элемента. Так, чтобы поместить элемент на расстоянии 100 пикселов от левого края и 100 пикселов от верхнего края документа, можно задать CSS-стили в атрибуте `style`, как показано ниже:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

Если для элемента задана абсолютная позиция, значения его свойств `top` и `left` интерпретируются как расстояния до ближайшего родительского элемента, свойство `position` которого имеет любое значение, кроме `static`. Если абсолютно позиционируемый элемент не имеет позиционируемого предка, его свойства `top` и `left` будут определять координаты относительно начала документа – левого верхнего его угла. Если вам потребуется позиционировать элемент относительно контейнера, который является частью обычного потока вывода документа, определите в контейнере свойство `position:relative` и укажите значение `0px` в свойствах `top` и `left` контейнера. В этом случае контейнер будет позиционироваться динамически и останется при этом на обычном месте в потоке вывода документа. Любые абсолютно позиционируемые вложенные элементы будут позиционироваться относительно контейнера.

При позиционировании элементов чаще всего задается положение верхнего левого угла элемента с помощью атрибутов `left` и `top`, но точно так же можно задать положение нижнего и правого краев элемента относительно нижнего и правого краев элемента-контейнера с помощью атрибутов `right` и `bottom`. Например, при помощи следующих стилей можно указать, что правый нижний угол элемента должен находиться в правом нижнем углу документа (предполагается, что он не вложен в другой динамический элемент):

```
position: absolute; right: 0px; bottom: 0px;
```

Чтобы верхний край элемента располагался в 10 пикселах от верхнего края окна, а правый – в 10 пикселах от правого края окна и при этом не прокручивался вместе с документом, можно использовать такие стили:

```
position: fixed; right: 10px; top: 10px;
```

Помимо позиций элементов CSS позволяет указывать их размеры. Чаще всего это делается путем задания значений свойств стиля `width` и `height`. Например, следующая разметка HTML создаст абсолютно позиционированный элемент без содержимого. Его свойствам `width`, `height` и `background-color` присвоены такие значения, что он будет отображаться в виде маленького синего квадрата:

```
<div style="position: absolute; top: 10px; left: 10px;
            width: 10px; height: 10px; background-color: blue">
</div>
```

Другой способ задать ширину элемента состоит в одновременном задании значений обоих свойств, `left` и `right`. Аналогично можно задать высоту элемента, одновременно указав оба свойства, `top` и `bottom`. Однако если задать значения для свойств `left`, `right` и `width`, то свойство `width` переопределит свойство `right`, а если ограничивается высота элемента, то более высоким приоритетом будет пользоваться свойство `height`.

Имейте в виду, что задавать размер каждого динамического элемента не обязательно. Некоторые элементы, такие как изображения, имеют собственный размер. Кроме того, для динамических элементов, включающих текст или другое потоковое содержимое, часто бывает достаточно указать желаемую ширину элемента и разрешить автоматическое определение высоты в зависимости от размещения содержимого элемента.

Стандарт CSS требует, чтобы в значениях свойств, определяющих позицию и размер, указывались единицы измерения. В предыдущих примерах значения свойств позиционирования и размера задавались с суффиксом «px», означающим «pixels» (пикселы). Можно также использовать другие единицы измерения: дюймы («in»), сантиметры («cm»), пункты («pt») и «em» (размер высоты строки текущего шрифта).

Помимо единиц измерения, представленных выше, CSS позволяет задавать положение и размер элемента в процентах от размера элемента-контейнера. Например, следующая разметка HTML создаст пустой элемент с черной рамкой, имеющий ширину и высоту в половину элемента-контейнера (или окна броузера) и расположенный в этом элементе по центру:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
            border: 2px solid black">
</div>
```

16.2.1.1. Третье измерение: z-index

Мы видели, что свойства `left`, `top`, `right` и `bottom` позволяют указывать координаты X и Y элемента в двухмерной плоскости элемента-контейнера. Свойство `z-index` определяет что-то вроде третьего измерения: оно позволяет определить порядок наложения элементов и указать, какой из двух или более перекрывающихся элементов должен располагаться поверх других. Атрибут `z-index` представляет собой целое число. По умолчанию его значение равно нулю, но можно задавать положительные и отрицательные значения. Когда два или более элементов перекрываются, они отображаются в порядке от меньших значений `z-index` к большим, т. е. элемент с большим значением `z-index` отображается поверх всех остальных. Если перекрывающиеся элементы имеют одинаковые значения `z-index`, они отображаются в порядке следования в документе, поэтому наверху оказывается последний из перекрывающихся элементов.

Обратите внимание, что порядок наложения определяется свойством z-index только для смежных элементов (т. е. для дочерних элементов одного контейнера). Если перекрываются два несмежных элемента, то на основе индивидуальных значений свойств z-index нельзя указать, какой из них должен находиться сверху. Вместо этого надо задать атрибут z-index для двух смежных контейнеров перекрывающихся элементов.

Способ размещения непозиционируемых элементов (элементов со значением по умолчанию режима позиционирования position:static) исключает возможность перекрытия, поэтому к ним свойство z-index не применяется. Тем не менее они получают значение z-index, по умолчанию равное нулю, т. е. позиционируемые элементы с положительным значением z-index отображаются поверх обычного потока вывода документа, а позиционируемые элементы с отрицательным значением z-index оказываются ниже обычного потока вывода документа.

16.2.1.2. Пример позиционирования средствами CSS: текст с тенью

Спецификация CSS3 включает свойство text-shadow, позволяющее добиться эффекта отбрасывания тени текстовыми элементами. Данное свойство поддерживается многими текущими броузерами, однако добиться эффекта тени можно и с помощью CSS-свойств позиционирования, для чего достаточно продублировать, сместив, выводимый текст:

```
<!-- Свойство text-shadow производит тень автоматически -->
<span style="text-shadow: 3px 3px 1px #888">С тенью</span>

<!--
Ниже показано, как добиться похожего эффекта с помощью механизма позиционирования.
-->
<span style="position:relative;">
  С тенью     <!-- Это основной текст, отбрасывающий тень. -->
  <span style="position:absolute; top:3px; left:3px; z-index:-1; color: #888">
    С тенью   <!-- Это тень -->
  </span>
</span>
```

Текст, отбрасывающий тень, заключается в относительно позиционируемый элемент <span>. Для него не определяются свойства, задающие позицию, поэтому текст отображается в обычной позиции в потоке. Тень заключается в абсолютно позиционируемый элемент <span>, помещенный в относительно позиционируемый элемент <span> (и поэтому позиционируется относительно него). Свойство z-index обеспечивает отображение тени под текстом.

16.2.2. Рамки, поля и отступы

Стандарт CSS позволяет задавать поля, рамки и отступы для любого элемента. Рамка элемента – это прямоугольник, обрисованный вокруг (полностью или частично) этого элемента. CSS-свойства позволяют задать стиль, цвет и толщину рамки:

```
border: solid black 1px; /* рамка рисуется черной сплошной линей,
                         /* толщиной 1 пиксел */
border: 3px dotted red;  /* рамка рисуется красной пунктирной линией */
                         /* толщиной 3 пиксела */
```

Толщину, стиль и цвет рамки можно задать с помощью отдельных CSS-свойств, а также отдельно для каждой из сторон элемента. Например, чтобы нарисовать линию под элементом, достаточно просто определить свойство `border-bottom`. Можно даже задать толщину, стиль и цвет рамки только для одной стороны элемента с помощью таких свойств, как `border-top-width` и `border-left-color`.

В CSS3 можно закруглить все углы рамки, определив свойство `border-radius`, или отдельные углы, задействовав более конкретные свойства. Например:

```
border-top-right-radius: 50px;
```

Свойства `margin` и `padding` задают ширину пустого пространства вокруг элемента. Отличие (очень важное) заключается в том, что свойство `margin` задает ширину пустого пространства снаружи от рамки, между рамкой и окружающими элементами, а свойство `padding` – внутри рамки, между рамкой и содержимым элемента. Поля (margin) позволяют визуально отделить элемент (возможно, окруженный рамкой) от соседних элементов в нормальном потоке вывода документа. Отступы (padding) позволяют визуально отделить содержимое элемента от его рамки. Если элемент не имеет рамки, определять ширину отступов обычно не требуется. Если элемент позиционируется динамически, он выпадает из нормального потока вывода документа, и для него не имеет смысла определять ширину полей.

Поля и отступы элемента задаются с помощью свойств `margin` и `padding`:

```
margin: 5px; padding: 5px;
```

Можно также определить поля и отступы для каждой из сторон элемента в отдельности:

```
margin-left: 25px;
padding-bottom: 5px;
```

Также можно задать величины полей и отступов для каждой из четырех сторон элемента с помощью свойств `margin` и `padding`, указав первым значение для верхней стороны и далее по часовой стрелке: сверху, справа, снизу и слева. Например, в следующем фрагменте приводятся два эквивалентных способа задания различных значений отступов для каждой из четырех сторон элемента:

```
padding: 1px 2px 3px 4px;
/* Предыдущая строка эквивалентна четырем следующим. */
padding-top: 1px;
padding-right: 2px;
padding-bottom: 3px;
padding-left: 4px;
```

Порядок работы с атрибутом `margin` ничем не отличается.

16.2.3. Блочная модель и детали позиционирования в CSS

Свойства стиля `margin`, `border` и `padding`, описанные выше, не относятся к числу свойств, которыми приходится часто управлять в сценариях. Причина, по которой они были описаны здесь, состоит в том, что поля, рамки и отступы являются частью *блочной модели* CSS, знать которую необходимо, чтобы до конца разобраться, как действуют CSS-свойства позиционирования.

Рис. 16.2 иллюстрирует блочную модель CSS и наглядно показывает, какой смысл имеют свойства `top`, `left`, `width` и `height` для элементов с рамками и отступами.

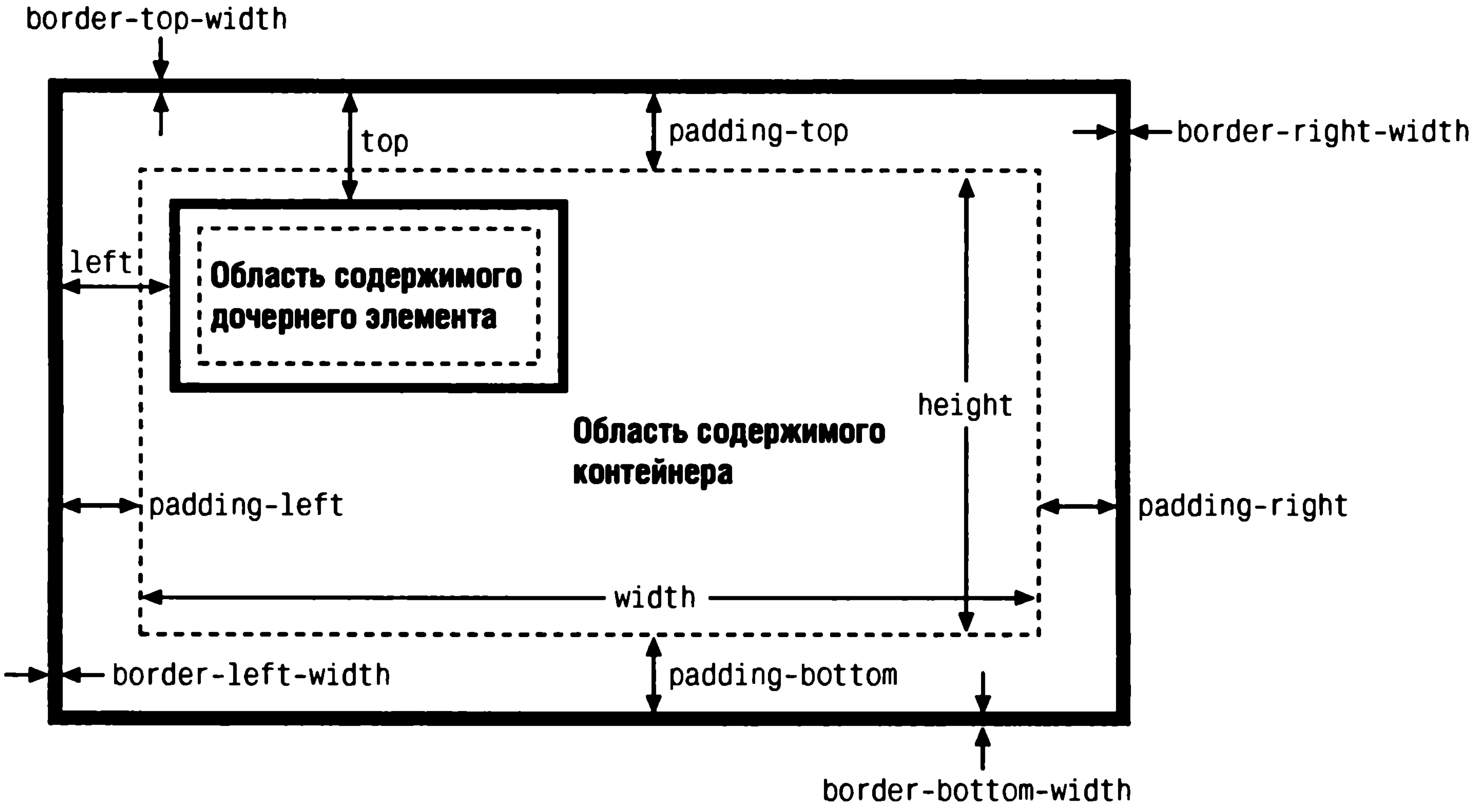

border-top-width
top
padding-top
border-right-width
left
Область содержимого дочернего элемента
height
Область содержимого контейнера
padding-left
padding-right
width
border-left-width
padding-bottom
border-bottom-width

Рис. 16.2. Блочная модель в CSS: рамки, отступы и свойства позиционирования

На рис. 16.2 показан абсолютно позиционируемый элемент, вложенный в позиционируемый элемент-контейнер. Оба элемента, контейнер и вложенный в него, имеют рамки и отступы, и на рисунке показаны CSS-свойства, определяющие ширину отступов и толщину рамок для каждой стороны элемента-контейнера. Обратите внимание, что здесь не показаны свойства `margin`: поля не используются при отображении абсолютно позиционируемых элементов.

Рис. 16.2 содержит также другую важную информацию. Во-первых, свойства `width` и `height` задают только размеры области содержимого элемента – они не включают дополнительное пространство, занимаемое отступами или рамками (и полями) элементов. Чтобы определить полный размер, занимаемый на экране элементом с рамкой, необходимо прибавить к ширине элемента ширину левого и правого отступов и толщину левой и правой рамок, а к высоте элемента необходимо прибавить ширину верхнего и нижнего отступов и толщину верхней и нижней рамок.

Во-вторых, свойства `left` и `top` задают расстояние от внутреннего края рамки контейнера до внешнего края рамки позиционируемого элемента. Эти свойства определяют расстояние не от левого верхнего угла области содержимого контейнера, а от левого верхнего угла области, занимаемой отступами контейнера. Аналогично свойства `right` и `bottom` определяют расстояние от правого нижнего угла области, занимаемой отступами.

Для полной ясности рассмотрим несложный пример. Предположим, что у вас есть динамически позиционируемый элемент, вокруг содержимого которого имеются отступы размером 10 пикселов, а вокруг них – рамка толщиной 5 пикселов. Теперь предположим, что вы динамически позиционируете дочерний элемент внутри этого контейнера. Если установить свойство `left` дочернего элемента равным «0px», обнаружится, что левый край дочернего элемента будет находиться непосредственно у внутреннего края рамки контейнера. При этом значении свойства дочерний элемент перекроет отступы контейнера, хотя предполагается, что они остаются пустыми (т. к. для этого и предназначены отступы). Чтобы поместить дочерний элемент в левый верхний угол области содержимого контейнера, необходимо присвоить свойствам `left` и `top` значение «10px».

16.2.3.1. Модель border-box и свойство box-sizing

Стандартная блочная модель CSS определяет, что свойства стиля `width` и `height` задают размер области содержимого элемента и не учитывают ширину отступов и толщину рамки. Эту модель можно было бы назвать «content-box». Эта модель имеет исключения, наблюдаемые в старых версиях IE и в новых версиях CSS. При отображении страниц в IE версии ниже 6 или в IE6, 7 или 8 в «режиме совместимости» (когда страница не имеет объявления `<!DOCTYPE>` или это объявление недостаточно строгое), свойства `width` и `height` включают ширину отступов и толщину рамок.

Поведение IE ошибочно, но нестандартная блочная модель, реализованная в IE, иногда бывает весьма удобна. Учитывая это, в стандарт CSS3 было добавлено свойство `box-sizing`. По умолчанию оно имеет значение `content-box`, которое указывает, что используется стандартная блочная модель, описанная выше. Если вместо него указать значение `box-sizing: border-box`, броузер будет использовать блочную модель IE и свойства `width` и `height` будут включать рамки и отступы. Модель `border-box` особенно удобно использовать, когда желательно задать общий размер элемента в процентах, а ширину отступов и толщину рамки – в пикселах:

```
<div style="box-sizing:border-box; width: 50%;
            padding: 10px; border: solid black 2px;">
```

Свойство `box-sizing` поддерживается всеми текущими броузерами, но пока не во всех оно реализовано без префикса. В Chrome и Safari свойство имеет имя `-webkit-box-sizing`. В Firefox – `-moz-box-sizing`. В Opera и IE8 (и выше) свойство имеет имя `box-sizing` без префикса.

Будущей альтернативой модели `border-box`, предусматриваемой стандартом CSS3, являются вычисляемые значения свойств:

```
<div style="width: calc(50%-12px); padding: 10px; border: solid black 2px;">
```

Возможность вычисления значений CSS-свойств с применением `calc()` поддерживается в IE9 и в Firefox 4 (как `-moz-calc()`).

16.2.4. Отображение и видимость элементов

Управлять видимостью элемента документа позволяют два CSS-свойства: `visibility` и `display`. Пользоваться свойством `visibility` очень просто: если оно имеет значение `hidden`, то элемент не отображается, если `visible`, – отображается. Свойство `display` является более универсальным и служит для задания варианта отображения элемента, определяя, блочный это элемент, встраиваемый, списочный или какой-нибудь другой. Если же свойство `display` имеет значение `none`, то элемент вообще не отображается и для него даже не выделяется место на странице.

Различие между свойствами стиля `visibility` и `display` имеет отношение к их воздействию на статически или относительно позиционируемые элементы. Для элемента, расположенного в нормальном потоке вывода документа, установка свойства `visibility` в значение `hidden` делает элемент невидимым, но резервирует для него место в документе. Такой элемент может повторно скрываться и отображаться без изменения компоновки документа. Однако если свойство `display` элемента установлено в значение `none`, то место в документе для него не выделяется; элементы с обеих сторон от него смыкаются, как будто его вообще не существует. Свойство `display` может пригодиться, например, при создании разворачивающихся и сворачивающихся списков.

Обратите внимание, что нет особого смысла использовать атрибуты `visibility` и `display`, чтобы сделать элемент невидимым, если вы не собираетесь динамически устанавливать их в сценарии на языке JavaScript, чтобы в какой-то момент сделать его снова видимым! Как это делается, будет показано далее в этой главе.

16.2.5. Цвет, прозрачность и полупрозрачность

Цвет текста, содержащегося в элементе документа, можно задать с помощью CSS-свойства `color`. Цвет фона любого элемента – с помощью свойства `background-color`. Выше мы видели, что цвет рамки элемента можно задать с помощью свойства `border-color` или сокращенной формы его написания `border`.

При обсуждении рамок мы рассматривали пример, в котором цвет рамки задавался указанием названий наиболее распространенных цветов, таких как «red» (красный) и «black» (черный). Стандарт CSS поддерживает множество таких обозначений цветов на английском языке, но имеется более универсальный способ определения цветов, который заключается в использовании шестнадцатеричных цифр, определяющих красную, зеленую и синюю составляющие цвета. Значения каждой из составляющих могут задаваться одной или двумя цифрами. Например:

```
#000000 /* черный */
#fff    /* белый */
#f00    /* ярко-красный */
#404080 /* ненасыщенный темно-синий */
#ccc    /* светло-серый */
```

Стандарт CSS3 дополнительно определяет возможность определения цвета в формате RGBA (значения красной, зеленой и синей составляющих плюс *альфа*-значение, определяющее степень прозрачности). Формат RGBA поддерживается всеми современными броузерами, кроме IE; ожидается, однако, что его поддержка будет включена в IE9. CSS3 также определяет поддержку форматов HSL (hue-saturation-value – тон-насыщенность-яркость) и HSLA. И снова эти форматы поддерживаются броузерами Firefox, Safari и Chrome, но не поддерживаются IE.

Таблицы CSS позволяют точно указать позицию, размеры, цвета фона и рамки элемента, что обеспечивает элементарные графические средства рисования прямоугольников и (если до предела уменьшить высоту или ширину) горизонтальных или вертикальных линий. В предыдущее издание книги был включен пример рисования столбчатых диаграмм средствами CSS, но в этом издании он был заменен расширенным описанием элемента `<canvas>`. (Подробнее о работе с графикой на стороне клиента рассказывается в главе 21.)

В дополнение к атрибуту `background-color` можно также указать изображение, которое должно использоваться в качестве фоновой картинки элемента. Свойство `background-image` определяет фоновое изображение, а свойства `background-attachment`, `background-position` и `background-repeat` уточняют некоторые параметры рисования изображения. Сокращенная форма записи – свойство `background`, позволяющее указывать все эти атрибуты вместе. Свойства, определяющие фоновый рисунок, могут применяться для создания довольно интересных визуальных эффектов, но это уже выходит за рамки темы данной книги.

Очень важно понимать, что если цвет фона или фоновый рисунок элемента не задан, то фон элемента обычно прозрачный. Например, если поверх некоторого текста в обычном потоке вывода документа расположить элемент `<div>` с абсолютным позиционированием, то по умолчанию текст будет виден через элемент `<div>`. Если же элемент `<div>` содержит собственный текст, символы окажутся наложенными друг на друга, образуя трудную для чтения мешанину. Однако не все элементы по умолчанию прозрачны. Например, элементы форм выглядели бы нелепо с прозрачным фоном, и такие элементы, как `<button>`, имеют цвет фона по умолчанию. Переопределить значение цвета по умолчанию можно с помощью свойства `background-color`; при необходимости можно явно установить цвет фона прозрачным («transparent»).

Прозрачность, о которой мы до сих пор говорили, может быть либо полной, либо нулевой: элемент имеет либо прозрачный, либо непрозрачный фон. Однако существует возможность получить полупрозрачный элемент (для содержимого как заднего, так и переднего плана). (Пример полупрозрачного элемента приведен на рис. 16.3.) Делается это с помощью свойства `opacity`, определяемого стандартом CSS3. Значением этого свойства является число в диапазоне от 0 до 1, где 1 означает 100-процентную непрозрачность (значение по умолчанию), а 0 – 100-процентную прозрачность. Свойство `opacity` поддерживается всеми текущими броузерами, кроме IE. В IE аналогичная функциональность реализуется с помощью специфичного свойства `filter`. Чтобы сделать элемент непрозрачным на 75%, можно воспользоваться следующими CSS-стилями:

```
opacity: .75;               /* стандартный стиль прозрачности в CSS3 */
filter: alpha(opacity=75); /* прозрачность в IE; обратите внимание */
                            /* на отсутствие десятичной точки */
```

16.2.6. Частичная видимость: свойства overflow и clip

Свойство `visibility` позволяет полностью скрыть элемент документа. С помощью свойств `overflow` и `clip` можно отобразить только часть элемента. Свойство `overflow` определяет, что происходит, когда содержимое документа превышает размер, указанный для элемента (например, в свойствах стиля `width` и `height`). Далее перечислены допустимые значения этого свойства и указано их назначение:

`visible`

Содержимое может выходить за пределы и по необходимости прорисовываться вне прямоугольника элемента. Это значение по умолчанию.

`hidden`

Содержимое, вышедшее за пределы элемента, обрезается и скрывается, так что никакая часть содержимого никогда не прорисовывается вне области, определяемой свойствами размера и позиционирования.

`scroll`

Область элемента имеет постоянные горизонтальную и вертикальную полосы прокрутки. Если содержимое превышает размеры области, полосы прокрутки позволяют увидеть остальное содержимое. Это значение учитывается, только когда документ отображается на экране компьютера; когда документ выводится, например, на бумагу, полосы прокрутки, очевидно, не имеют смысла.

`auto`

Полосы прокрутки отображаются не постоянно, а только когда содержимое превышает размер элемента.

Если свойство `overflow` определяет, что должно происходить, когда содержимое элемента превысит область элемента, то с помощью свойства `clip` можно точно указать, какая часть элемента должна отображаться независимо от того, выходит ли содержимое за пределы элемента. Это свойство особенно полезно для создания эффектов, когда элемент открывается или проявляется постепенно.

Значение свойства `clip` задает область отсечения элемента. В CSS2 области отсечения прямоугольные, но синтаксис атрибута `clip` обеспечивает возможность в следующих версиях стандарта задавать области отсечения, отличные от прямоугольных. Синтаксис свойства `clip`:

```
rect(top right bottom left)
```

Значения *`top`*, *`right`*, *`bottom`* и *`left`* задают границы прямоугольника отсечения относительно левого верхнего угла области элемента. Чтобы, например, вывести только часть элемента в области 100×100 пикселов, можно задать для этого элемента следующий атрибут `style`:

```
style="clip: rect(0px 100px 100px 0px);"
```

Обратите внимание, что четыре значения в скобках представляют собой значения длины и должны включать спецификацию единиц измерения, например `px` для пикселов. Проценты здесь не допускаются. Значения могут быть отрицательными – это будет означать, что область отсечения выходит за переделы области, определенной для элемента. Для любого из четырех значений ключевое слово `auto` указывает, что этот край области отсечения совпадает с соответствующим краем самого элемента. Например, можно вывести только левые 100 пикселов элемента с помощью следующего атрибута `style`:

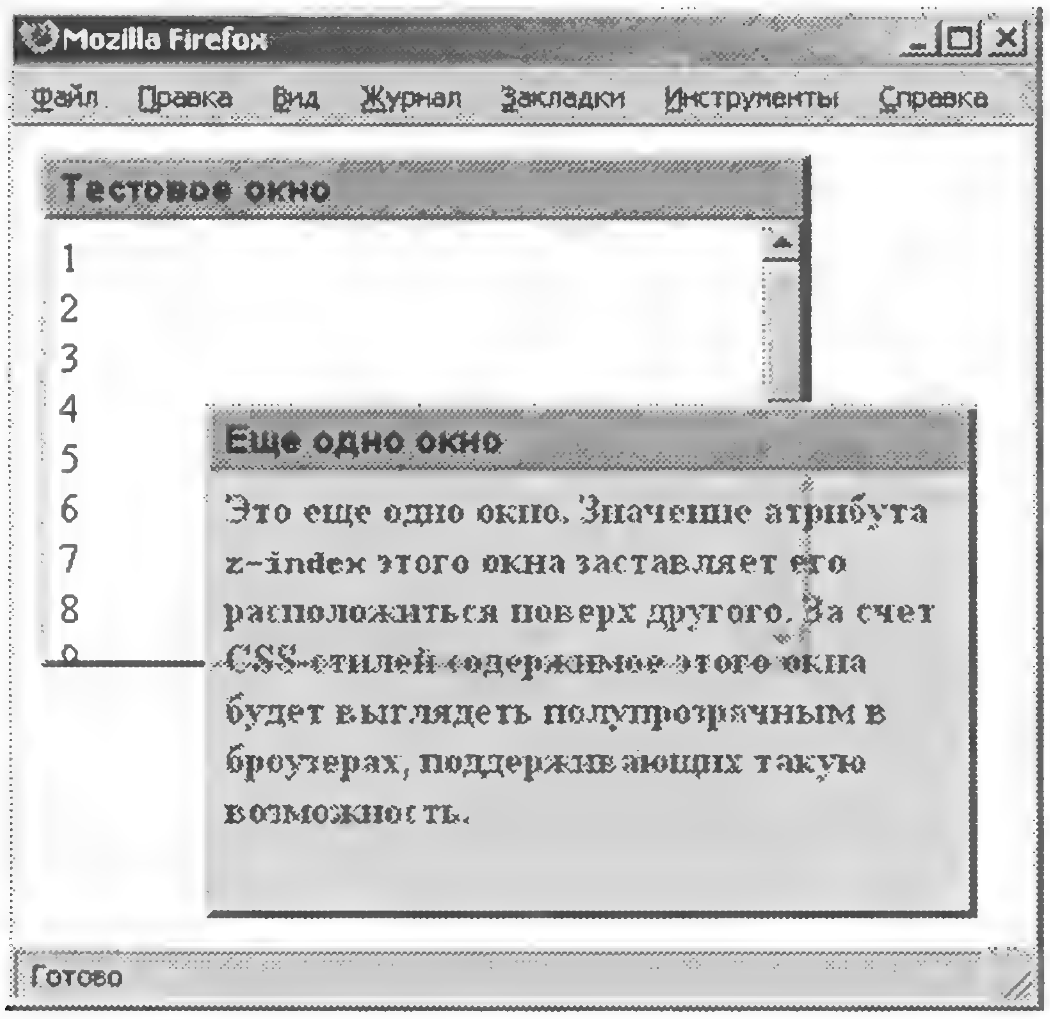
```
style="clip: rect(auto 100px auto auto);"
```

Обратите внимание, что между значениями нет запятых, и края области отсечения задаются по часовой стрелке, начиная с верхнего края.

16.2.7. Пример: перекрытие полупрозрачных окон

Данный раздел завершается примером, который демонстрирует порядок работы с большинством обсуждавшихся CSS-свойств. В примере 16.2 CSS-стили исполь-

зуются для создания визуального эффекта наложения полупрозрачного окна на другое окно, обладающее полосой прокрутки. Результат приводится на рис. 16.3.



Это еще одно окно. Значение атрибута z-index этого окна заставляет его расположиться поверх другого. За счет CSS-стилей содержимое этого окна будет выглядеть полупрозрачным в броузерах, поддерживающих такую возможность.

Рис. 16.3. Окна, созданные с помощью CSS

Пример не содержит JavaScript-код и в нем нет никаких обработчиков событий, поэтому возможность взаимодействия с окнами отсутствует (иначе как через полосу прокрутки), но это очень интересная демонстрация эффектов, которые можно получить средствами CSS.

Пример 16.2. Отображение окон с использованием CSS-стилей

```
<!DOCTYPE html">
<head>
<style type="text/css">
/**
 * Эта таблица CSS-стилей определяет три правила стилей, которые используются
 * в теле документа для создания визуального эффекта "окна". В правилах использованы
 * свойства позиционирования для установки общего размера окна и расположения
 * его компонентов. Изменение размеров окна требует аккуратного
 * изменения атрибутов позиционирования во всех трех правилах.
**/
div.window {                      /* Определяет размер и рамку окна */
    position: absolute;           /* Положение задается в другом месте */
    width: 300px; height: 200px;  /* Размер окна без учета рамок */
    border: 3px outset gray;      /* Обратите внимание на 3D-эффект рамки */
}

div.titlebar {                    /* Задает положение, размер и стиль заголовка */
    position: absolute;           /* Это позиционируемый элемент */
    top: 0px; height: 18px;       /* Высота заголовка 18px + отступ и рамка */
    width: 290px;                 /* 290 + 5px отступы слева и справа = 300 */
    background-color: #aaa;       /* Цвет заголовка */
    border-bottom: groove gray 2px; /* Заголовок имеет рамку только снизу */
    padding: 3px 5px 2px 5px;       /* Значения по часовой стрелке:
                                    /* сверху, справа, снизу, слева */
    font: bold 11pt sans-serif;     /* Шрифт заголовка */
}

div.content {      /* Задает размер, положение и прокрутку содержимого окна */
    position: absolute;             /* Это позиционируемый элемент */
    top: 25px;                      /* 18px заголовок+2px рамка+3px+2px отступ */
    height: 165px;                  /* 200px всего - 25px заголовок - 10px отступ */
    width: 290px;                   /* 300px ширина - 10px отступ */
    padding: 5px;                   /* Отступы со всех четырех сторон */
    overflow: auto;                 /* Разрешить появление полос прокрутки */
    background-color: #ffffff;      /* По умолчанию белый фон */
}

div.translucent {                   /* Этот класс делает окно частично прозрачным */
    opacity: .75;                   /* Стандартный стиль прозрачности */
    filter: alpha(opacity=75);      /* Прозрачность для IE */
}
</style>
</head>

<body>
<!-- Порядок определения окна: элемент div "окна" с заголовком и элемент div -->
<!-- с содержимым, вложенный между ними. Обратите внимание, как задается -->
<!-- позиционирование с помощью атрибута style, дополняющего -->
<!-- стили из таблицы стилей -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Тестовое окно</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- Множество строк для -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br><!-- демонстр. прокрутки -->
</div>
</div>

<!-- Это еще одно окно с другими позицией, цветом и шрифтом -->
<div class="window" style="left: 75px; top: 110px; z-index: 20;">
<div class="titlebar">Еще одно окно</div>
<div class="content translucent"
     style="background-color:#ccc; font-weight:bold;">
Это еще одно окно. Значение атрибута <tt>z-index</tt> этого окна заставляет
его расположиться поверх другого. За счет CSS-стилей содержимое этого окна
будет выглядеть полупрозрачным в броузерах, поддерживающих такую возможность.
</div>
</div>
</body>
```

Основной недостаток этого примера в том, что таблица стилей задает фиксированный размер всех окон. Так как заголовок и содержимое окна должны точно позиционироваться внутри окна, изменение размера окна требует изменения значений различных свойств позиционирования во всех трех правилах, определенных в таблице стилей. Это трудно сделать в статическом HTML-документе, но все становится проще, если использовать сценарий, устанавливающий все необходимые свойства. Эта возможность рассматривается в следующем разделе.

16.3. Управление встроенными стилями

Самый простой способ управления стилями CSS – это манипулирование атрибутом `style` отдельных элементов документа. Как и для большинства HTML-атрибутов, атрибуту `style` соответствует одноименное свойство объекта `Element`, и им можно манипулировать в сценариях на языке JavaScript. Однако свойство `style` имеет одну отличительную особенность: его значением является не строка, а объект `CSSStyleDeclaration`. Свойства этого объекта представляют CSS-свойства, определенные в HTML-атрибуте `style`. Например, чтобы вывести содержимое текстового элемента `e` крупным, полужирным шрифтом синего цвета, можно выполнить следующие операции для записи желаемых значений в свойства, которые соответствуют свойствам стиля `font-size`, `font-weight` и `color`:

```
e.style.fontSize = "24pt";
e.style.fontWeight = "bold";
e.style.color = "blue";
```

При работе со свойствами стиля объекта `CSSStyleDeclaration` не забывайте, что все значения должны задаваться в виде строк. В таблице стилей или атрибуте `style` можно написать:

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

Чтобы сделать то же самое для элемента `e` в JavaScript, необходимо заключить все значения в кавычки:

```
e.style.position = "absolute";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Обратите внимание, что точки с запятыми не входят в строковые значения. Это точки с запятой, употребляемые в синтаксисе языка JavaScript. Точки с запятой, используемые в таблицах стилей CSS, не нужны в строковых значениях, устанавливаемых с помощью JavaScript.

Кроме того, помните, что во всех свойствах позиционирования должны быть указаны единицы измерения. То есть нельзя устанавливать свойство `left` подобным образом:

```
e.style.left = 300;   // Неправильно: это число, а не строка
e.style.left = "300"; // Неправильно: отсутствуют единицы измерения
```

Единицы измерения обязательны при установке свойств стиля в JavaScript – так же, как при установке свойств стиля в таблицах стилей. Ниже приводится правильный способ установки значения свойства `left` элемента `e`, равным 300 пикселам:

```
e.style.left = "300px";
```

Чтобы установить свойство `left` равным вычисляемому значению, обязательно добавьте единицы измерения в конце вычислений:

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

Как побочный эффект, добавление строки с единицами измерения преобразует вычисленное значение из числа в строку.

Соглашения об именах: CSS-свойства в JavaScript

Многие CSS-свойства стиля, такие как `font-size`, содержат в своих именах дефис. В языке JavaScript дефис интерпретируется как знак минус, поэтому нельзя записать выражение, приведенное ниже:

```
e.style.font-size = "24pt"; // Синтаксическая ошибка!
```

Таким образом, имена свойств объекта `CSSStyleDeclaration` слегка отличаются от имен реальных CSS-свойств. Если имя CSS-свойства содержит дефисы, имя свойства объекта `CSSStyleDeclaration` образуется путем удаления дефисов и перевода в верхний регистр буквы, непосредственно следующей за каждым из них. Другими словами, CSS-свойство `border-left-width` доступно через свойство `borderLeftWidth`, а к CSS-свойству `font-family` можно обратиться следующим образом:

```
e.style.fontFamily = "sans-serif";
```

Кроме того, когда CSS-свойство, такое как `float`, имеет имя, совпадающее с зарезервированным словом языка JavaScript, к этому имени добавляется префикс «css», чтобы создать допустимое имя свойства объекта `CSSStyleDeclaration`. То есть, чтобы прочитать или изменить значение CSS-свойства `float` элемента, следует использовать свойство `cssFloat` объекта `CSSStyleDeclaration`.

Напомню, что некоторые CSS-свойства, такие как `margin`, представляют собой сокращенную форму записи других свойств, таких как `margin-top`, `margin-right`, `margin-bottom` и `margin-left`. Объект `CSSStyleDeclaration` имеет свойства, соответствующие этим сокращенным формам записи свойств. Например, свойство `margin` можно установить следующим образом:

```
e.style.margin = topMargin + "px " + rightMargin + "px " +
                 bottomMargin + "px " + leftMargin + "px";
```

Хотя, возможно, кому-то будет проще установить четыре свойства полей по отдельности:

```
e.style.marginTop = topMargin + "px";
e.style.marginRight = rightMargin + "px";
e.style.marginBottom = bottomMargin + "px";
e.style.marginLeft = leftMargin + "px";
```

Атрибут `style` HTML-элемента – это его *встроенный стиль*, и он переопределяет любые правила стилей в таблице CSS. Встроенные стили в целом удобно использовать для установки значений стиля, и именно такой подход использовался во всех примерах выше. Сценарии могут читать свойства объекта `CSSStyleDeclaration`, представляющего встроенные стили, но они возвращают осмысленные значения, только если были ранее установлены сценарием на языке JavaScript или если HTML-элемент имеет встроенный атрибут `style`, установивший нужные свойства. Например, документ может включать таблицу стилей, устанавливающую левое

поле для всех абзацев равным 30 пикселам, но если прочитать свойство `leftMargin` одного из этих элементов, будет получена пустая строка, если только этот абзац не имеет атрибут `style`, переопределяющий значение, установленное таблицей стилей.

Чтение встроенного стиля элемента представляет особую сложность, когда выполняется чтение свойств стиля, имеющих единицы измерения, а также свойств сокращенной формы записи: сценарий должен включать далеко не простую реализацию синтаксического анализа строк с CSS-стилями, чтобы обеспечить возможность извлечения и дальнейшего использования значений. В целом, встроенный стиль элемента удобно использовать только для установки стилей. Если сценарию потребуется получить стиль элемента, лучше использовать вычисленные стили, которые обсуждаются в разделе 16.4.

Иногда бывает проще прочитать или записать единственную строку во встроенный стиль элемента, чем обращаться к объекту `CSSStyleDeclaration`. Для этого можно использовать методы `getAttribute()` и `setAttribute()` объекта `Element` или свойство `cssText` объекта `CSSStyleDeclaration`:

```
// Обе инструкции, следующие ниже, записывают в атрибут style
// элемента e строку s:
e.setAttribute("style", s);
e.style.cssText = s;

// Обе инструкции, следующие ниже, получают значение атрибута style
// элемента e в виде строки:
s = e.getAttribute("style");
s = e.style.cssText;
```

16.3.1. Создание анимационных эффектов средствами CSS

Одной из наиболее типичных областей применения CSS является воспроизведение визуальных анимационных эффектов. Реализовать их можно с помощью метода `setTimeout()` или `setInterval()` (раздел 14.1), используя их для организации многократных вызовов функции, изменяющей встроенный стиль элемента. Пример 16.3 демонстрирует две такие функции, `shake()` и `fadeOut()`. Функция `shake()` перемещает, или «встряхивает» (shakes), элемент из стороны в сторону. Ее можно использовать, например, для привлечения внимания пользователя в случае ввода некорректных данных. Функция `fadeOut()` уменьшает непрозрачность элемента в течение указанного периода времени (по умолчанию 500 миллисекунд), вызывая эффект его растворения до полного исчезновения.

Пример 16.3. Воспроизведение анимационных эффектов средствами CSS

```
// Делает элемент e относительно позиционируемым и перемещает его влево и вправо.
// Первым аргументом может быть объект элемента или значение атрибута id требуемого
// элемента. Если во втором аргументе передать функцию, она будет вызвана с элементом e
// в виде аргумента по завершении воспроизведения анимации. Третий аргумент определяет
// величину смещения элемента e. По умолчанию принимает значение 5 пикселов.
// Четвертый аргумент определяет, как долго должен воспроизводиться эффект.
// По умолчанию эффект длится 500 мсек.
function shake(e, oncomplete, distance, time) {
    // Обработка аргументов
    if (typeof e === "string") e = document.getElementById(e);
    if (!time) time = 500;
    if (!distance) distance = 5;

    var originalStyle = e.style.cssText; // Сохранить оригинальный стиль e
    e.style.position = "relative";       // Сделать относит. позиционируемым
    var start = (new Date()).getTime();  // Запомнить момент начала анимации
    animate();                           // Запустить анимацию

    // Эта функция проверяет прошедшее время и изменяет координаты e.
    // Если анимацию пора завершать, восстанавливает первоначальное состояние
    // элемента e. Иначе изменяет координаты e и планирует следующий свой вызов.
    function animate() {
        var now = (new Date()).getTime(); // Получить текущее время
        var elapsed = now-start;          // Сколько прошло времени с начала?
        var fraction = elapsed/time;      // Доля от требуемого времени?

        if (fraction < 1) {               // Если рано завершать анимацию
            // Вычислить координату x элемента e как функцию от доли общего
            // времени анимации. Здесь используется синусоидальная функция,
            // а доля общего времени воспроизведения умножается на 4pi,
            // поэтому перемещение взад и вперед выполняется дважды.
            var x = distance * Math.sin(fraction*4*Math.PI);
            e.style.left = x + "px";

            // Попробовать вызвать себя через 25 мсек или в конце запланированного
            // отрезка общего времени воспроизведения. Мы стремимся сделать
            // анимацию гладкой, воспроизводя ее со скоростью 40 кадров/сек.
            setTimeout(animate, Math.min(25, time-elapsed));
        }
        else {                              // Иначе анимацию пора завершать
            e.style.cssText = originalStyle // Восстановить первонач. стиль
            if (oncomplete) oncomplete(e);  // Вызвать ф-цию обратного вызова
        }
    }
}

// Растворяет e от состояния полной непрозрачности до состояния полной прозрачности
// за указанное количество миллисекунд. Предполагается, что, когда вызывается
// эта функция, e полностью непрозрачен. oncomplete - необязательная функция,
// которая будет вызвана с элементом e в виде аргумента по завершении анимации.
// Если аргумент time не задан, устанавливается интервал 500 мсек.
// Эта функция не работает в IE, но ее можно модифицировать так, чтобы
// в дополнение к свойству opacity она использовала нестандартное
// свойство filter, реализованное в IE.
function fadeOut(e, oncomplete, time) {
    if (typeof e === "string") e = document.getElementById(e);
    if (!time) time = 500;

    // В качестве простой "функции перехода", чтобы сделать анимацию немного
    // нелинейной, используется Math.sqrt: сначала растворение идет быстро,
    // а затем несколько замедляется.
    var ease = Math.sqrt;

    var start = (new Date()).getTime(); // Запомнить момент начала анимации
```

```
        animate();                       // И запустить анимацию

        function animate() {
            var elapsed = (new Date()).getTime()-start; // Прошедшее время
            var fraction = elapsed/time;                // Доля от общего времени
            if (fraction < 1) {                         // Если не пора завершать
                var opacity = 1 - ease(fraction);       // Вычислить непрозрачн.
                e.style.opacity = String(opacity);      // Установить ее в e
                setTimeout(animate,                     // Запланировать очередной
                        Math.min(25, time-elapsed));    // кадр
            }
            else {                                      // Иначе завершить
                e.style.opacity = "0";         // Сделать e полностью прозрачным
                if (oncomplete) oncomplete(e); // Вызвать ф-цию обратного вызова
            }
        }
    }
```

Обе функции, `shake()` и `fadeOut()`, принимают необязательную функцию обратного вызова во втором аргументе. Если эта функция указана, она будет вызвана по завершении воспроизведения анимационного эффекта. Элемент, к которому применялся анимационный эффект, будет передан функции обратного вызова в виде аргумента. Следующая разметка HTML создает кнопку, для которой после щелчка на ней воспроизводится эффект встряхивания, а затем эффект растворения:

```
<button onclick="shake(this, fadeOut);">Встряхнуть и растворить</button>
```

Обратите внимание, насколько функции `shake()` и `fadeOut()` похожи друг на друга. Они обе могут служить шаблонами для реализации похожих анимационных эффектов с использованием CSS-свойств. Однако клиентские библиотеки, такие как jQuery, обычно поддерживают набор предопределенных визуальных эффектов, поэтому вам вообще может никогда не потребоваться писать собственные функции воспроизведения анимационных эффектов, такие как `shake()`, если только вам не понадобится создать какой-нибудь сложный визуальный эффект. Одной из первых и наиболее примечательных библиотек визуальных эффектов является библиотека Scriptaculous, которая предназначалась для работы в составе фреймворка Prototype. За дополнительной информацией обращайтесь по адресу *http://script.aculo.us/* и *http://scripty2.com/*.

Модуль «CSS3 Transitions» определяет еще один способ реализации анимационных эффектов с помощью таблиц стилей, полностью устраняющий необходимость писать программный код. Например, вместо функции `fadeOut()` можно было бы определить правило CSS, как показано ниже:

```
.fadeable { transition: opacity .5s ease-in }
```

Это правило говорит, что всякий раз, когда изменяется непрозрачность элемента с классом «fadeable», это изменение должно протекать плавно (от текущего до нового значения) в течение половины секунды с использованием нелинейной функции перехода. Модуль «CSS Transitions» еще не был стандартизован, но его положения уже реализованы в броузерах Safari и Chrome в виде свойства `-webkit-transition`. На момент написания этих строк в Firefox 4 также была добавлена поддержка свойства `-moz-transition`.

16.4. Вычисленные стили

Свойство `style` элемента определяет *встроенный* стиль элемента. Оно имеет преимущество перед всеми таблицами стилей и с успехом может применяться для установки CSS-свойств для изменения визуального представления элемента. Однако в общем случае к нему нет смысла обращаться, когда требуется узнать фактически примененные к элементу стили. То, что требуется в этом случае, называется *вычисленным стилем*. Вычисленный стиль элемента – это набор значений свойств, которые броузер получил (или вычислил) из встроенного стиля и всех правил из всех таблиц стилей, которые применяются к элементу: это набор свойств, фактически используемый при отображении элемента. Подобно встроенным стилям, вычисленные стили представлены объектом `CSSStyleDeclaration`. Однако в отличие от встроенных стилей, вычисленные стили доступны только для чтения. Эти стили нельзя изменить, но вычисленный объект `CSSStyleDeclaration` позволяет точно узнать значения свойств стилей, которые броузер использовал при отображении соответствующего элемента.

Получить вычисленный стиль элемента можно с помощью метода `getComputedStyle()` объекта `Window`. Первым аргументом этому методу передается элемент, вычисленный стиль которого требуется вернуть. Второй аргумент является обязательным, и в нем обычно передается значение `null` или пустая строка, но в нем также может передаваться строка с именем псевдоэлемента CSS, таким как «:before», «:after», «:first-line» или «:first-letter»:

```
var title = document.getElementById("section1title");
var titlestyles = window.getComputedStyle(element, null);
```

Возвращаемым значением метода `getComputedStyle()` является объект `CSSStyleDeclaration`, представляющий все стили, применяемые к указанному элементу (или псевдоэлементу). Между объектами `CSSStyleDeclaration`, представляющими встроенные стили и вычисленные стили, существует множество существенных отличий:

- Свойства вычисленного стиля доступны только для чтения.
- Свойства вычисленных стилей имеют абсолютные значения: относительные единицы измерения, такие как проценты и пункты, преобразуются в абсолютные значения. Любое свойство, которое определяет размер (например, ширина поля или размер шрифта) будет иметь значение, выраженное в пикселах. То есть его значением будет строка с суффиксом «px», поэтому вам необходимо будет реализовать ее синтаксический анализ, зато не придется беспокоиться об определении и преобразовании единиц измерений. Значения свойств, определяющих цвет, будут возвращаться в формате «rgb(#,#,#)» или «rgba(#,#,#,#)».
- Свойства, являющиеся краткой формой записи, не вычисляются – только фундаментальные свойства, на которых они основаны. Например, не следует пытаться получить значение свойства `margin`, вместо этого нужно обращаться к свойствам `marginLeft`, `marginTop` и т. д.
- Свойство `cssText` вычисленного стиля не определено.

Вычисленные и встроенные стили можно использовать совместно. В примере 16.4 определяются функции `scale()` и `scaleColor()`. Первая читает и анализирует вычисленный размер текста указанного элемента; вторая читает и анализирует вычисленный цвет фона элемента. Затем обе функции умножают полученное значение на заданное число и устанавливают результат, как встроенный стиль элемента. (Эти функции не работают в IE8 и в более ранних версиях: как вы узнаете далее, эти версии IE не поддерживают метод `getComputedStyle()`.)

Пример 16.4. Определение вычисленных стилей и установка встроенных стилей

```
// Умножает размер текста элемента e на указанное число factor
function scale(e, factor) {
    // Определить текущий размер текста, получив вычисленный стиль
    var size = parseInt(window.getComputedStyle(e,"").fontSize);
    // И использовать встроенный стиль, чтобы увеличить этот размер
    e.style.fontSize = factor*size + "px";
}

// Изменяет цвет фона элемента e, умножая компоненты цвета на указанное число.
// Значение factor > 1 осветляет цвет элемента, а factor < 1 затемняет его.
function scaleColor(e, factor) {
    var color = window.getComputedStyle(e, "").backgroundColor; // Получить
    var components = color.match(/[\d\.]+/g);  // Выбрать компоненты r,g,b и a
    for(var i = 0; i < 3; i++) {               // Цикл по r, g и b
        var x = Number(components[i]) * factor;// Умножить каждый из них
        x = Math.round(Math.min(Math.max(x, 0), 255)); // Округлить и установить границы
        components[i] = String(x);
    }
    if (components.length == 3)               // Цвет rgb()
        e.style.backgroundColor = "rgb(" + components.join() + ")";
    else                                      // Цвет rgba()
        e.style.backgroundColor = "rgba(" + components.join() + ")";
}
```

Работа с вычисленными стилями может оказаться весьма непростым делом, и обращение к ним не всегда возвращает ожидаемую информацию. Рассмотрим в качестве примера свойство `font-family`: оно принимает список разделенных запятыми имен семейств шрифтов для совместимости. При чтении свойства `fontFamily` вычисленного стиля вы ждете значение наиболее конкретного стиля `font-family`, применяемого к элементу. А в этом случае может вернуться такое значение, как «arial,helvetica,sans-serif», которое ничего не говорит о гарнитуре фактически используемого шрифта. Аналогично, если элемент не является абсолютно позиционируемым, при попытке определить его размеры или положение через свойства `top` и `left` вычисленного стиля часто возвращается значение «auto». Это вполне допустимое в CSS значение, но наверняка совсем не то, что вы ожидали бы получить.

Метод `getComputedStyle()` не поддерживается в IE8 и в более ранних версиях, но, как ожидается, он будет реализован в IE9. В IE все HTML-элементы имеют свойство `currentStyle`, значением которого является объект `CSSStyleDeclaration`. Свойство `currentStyle` в IE объединяет встроенные стили с таблицами стилей, но оно не является по-настоящему вычисленным стилем, потому что не преобразует относительные значения в абсолютные. При чтении свойств текущего стиля в IE могут возвращаться размеры в относительных единицах измерения, таких как «%» или «em», а цвета в виде неточных названий, таких как «red».

Стили CSS можно использовать, чтобы точно задать позицию и размер элемента документа, но чтение вычисленного стиля элемента является не самым лучшим способом узнать его размер и положение в документе. Более простые, переносимые решения приводятся в разделе 15.8.2.

16.5. CSS-классы

Альтернативой использованию отдельных CSS-стилей через свойство `style` является управление значением HTML-атрибута `class`. Изменение класса элемента изменяет набор селекторов стилей, применяемых к элементу, что может приводить к изменениям значений сразу нескольких CSS-свойств. Предположим, например, что вам требуется привлечь внимание пользователя к отдельным абзацам (или другим элементам) в документе. В этом случае можно было бы сначала определить особый стиль оформления для любых элементов с классом «attention»:

```
.attention { /* Стили для элементов, требующих внимания пользователя */
    background-color: yellow; /* Желтый фон */
    font-weight: bold;        /* Полужирный шрифт */
    border: solid black 2px;  /* Черная рамка */
}
```

Идентификатор `class` в языке JavaScript является зарезервированным словом, поэтому HTML-атрибут `class` в JavaScript-сценариях доступен в виде свойства с именем `className`. Ниже приводится пример, который устанавливает и очищает свойство `className` элемента, добавляя и убирая класс «attention»:

```
function grabAttention(e) { e.className = "attention"; }
function releaseAttention(e) { e.className = ""; }
```

HTML-элементы могут быть членами более чем одного CSS-класса – атрибут `class` может содержать целый список имен классов, разделенных пробелами. Имя `className` не совсем точно отражает смысл свойства: правильнее было бы дать ему имя `classNames`. Функции выше предполагают, что свойство `className` будет определять ноль или одно имя класса, и они непригодны в случаях, когда может использоваться более одного имени класса. Если элемент уже принадлежит некоторому классу, вызов функции `grabAttention()` для этого элемента затрет имя класса, присутствующее в свойстве `className`.

Стандарт HTML5 решает эту проблему, определяя свойство `classList` во всех элементах. Значением этого свойства является объект `DOMTokenList`: подобный массиву объект (раздел 7.11), доступный только для чтения, элементы которого содержат отдельные имена классов, присвоенных элементу. Однако самыми важными в нем являются не элементы массива, а методы `add()` и `remove()`, добавляющие и удаляющие отдельные имена классов из атрибута `class` элемента. Метод `toggle()` добавляет имя класса, если оно отсутствует, и удаляет в противном случае. Наконец, метод `contains()` проверяет присутствие указанного имени класса в атрибуте `class`.

Подобно другим классам коллекций в модели DOM, объект `DOMTokenList` является «живым» представлением множества классов в элементе, а не статическим слепком, который был действителен только в момент обращения к свойству `classList`. Если сценарий получит объект `DOMTokenList`, обратившись к свойству `classList` элемента, и затем изменит свойство `className` этого элемента, то выполненные изменения немедленно отразятся на полученном объекте `DOMTokenList`. Аналогично любые изменения, выполненные в объекте `DOMTokenList`, немедленно отразятся на значении свойства `className`.

На момент написания этих строк свойство `classList` не поддерживалось ни одним из текущих броузеров. Однако эту удобную функциональность легко можно реализовать самому, как показано в примере 16.5. Подобная реализация, позволяющая интерпретировать атрибут `class` элемента как множество имен классов, существенно упрощает выполнение многих задач, связанных с обработкой CSS.

Пример 16.5. classList(): интерпретирует className, как множество CSS-классов

```
/*
 * Возвращает свойство classList элемента e, если оно содержит один класс.
 * Иначе возвращает объект, имитирующий интерфейс DOMTokenList.
 * Возвращаемый объект имеет методы contains(), add(), remove(), toggle() и toString(),
 * позволяющие проверять и изменять набор классов элемента e. Если свойство classList
 * имеет встроенную поддержку в броузере, функция возвращает объект, подобный массиву,
 * который имеет свойство length и поддерживает возможность индексирования массива.
 * Имитация объекта DOMTokenList не подобна массиву, но имеет метод toArray(),
 * который возвращает истинный массив имен классов элемента.
 */
function classList(e) {
    if (e.classList) return e.classList; // Вернуть e.classList, если имеется
    else return new CSSClassList(e);     // Иначе попытаться подделать его
}

// CSSClassList - класс JavaScript, имитирующий объект DOMTokenList
function CSSClassList(e) { this.e = e; }

// Возвращает true, если e.className содержит класс c, иначе - false
CSSClassList.prototype.contains = function(c) {
    // Проверить, является ли c допустимым именем класса
    if (c.length === 0 || c.indexOf(" ") != -1)
        throw new Error("Недопустимое имя класса: '" + c + "'");
    // Сначала проверить общие случаи
    var classes = this.e.className;
    if (!classes) return false;      // e вообще не имеет классов
    if (classes === c) return true; // e имеет единственный класс,
                                     // имя которого точно совпадает с искомым

    // Иначе использовать RegExp для поиска c как отдельного слова
    // \b - в регулярных выражениях соответствует границе слова.
    return classes.search("\\b" + c + "\\b") != -1;
};

// Добавляет имя класса c в e.className, если оно отсутствует в списке
CSSClassList.prototype.add = function(c) {
    if (this.contains(c)) return; // Ничего не делать, если имя уже в списке
    var classes = this.e.className;
    if (classes && classes[classes.length-1] != " ")
        c = " " + c;             // Добавить пробел, если необходим
    this.e.className += c;       // Добавить имя c в className
};

// Удаляет все вхождения c из e.className
CSSClassList.prototype.remove = function(c) {
    // Убедиться, что c - допустимое имя класса
    if (c.length === 0 || c.indexOf(" ") != -1)
        throw new Error("Недопустимое имя класса: '" + c + "'");
    // Удалить все вхождения имени c как слова и все завершающие пробелы
    var pattern = new RegExp("\\b" + c + "\\b\\s*", "g");
    this.e.className = this.e.className.replace(pattern, "");
};

// Добавляет имя c в e.className, если оно отсутствует в списке, и возвращает true.
// Иначе удаляет все вхождения имени c из e.className и возвращает false.
CSSClassList.prototype.toggle = function(c) {
    if (this.contains(c)) { // Если e.className содержит c
        this.remove(c);     // удалить его.
        return false;
    }
    else {                  // Иначе:
        this.add(c);        // добавить его.
        return true;
    }
};

// Возвращает само значение e.className
CSSClassList.prototype.toString = function() { return this.e.className; };

// Возвращает имена из e.className
CSSClassList.prototype.toArray = function() {
    return this.e.className.match(/\b\w+\b/g) || [];
};
```

16.6. Управление таблицами стилей

До сих пор мы видели, как устанавливать и получать значения CSS-свойств стиля и классы отдельных элементов. Однако существует еще возможность управления самими таблицами стилей CSS. Обычно в этом не возникает необходимости, тем не менее такая возможность иногда бывает полезной, и в этом разделе коротко будут представленные возможные приемы.

При работе с самими таблицами стилей вам придется столкнуться с двумя типами объектов. Первый тип – это объекты `Element`, представляющие элементы `<style>` и `<link>`, которые содержат или ссылаются на таблицы стилей. Это обычные элементы документа, и если в них определить атрибут `id`, вы сможете выбирать их с помощью метода `document.getElementById()`. Второй тип объектов – объекты `CSSStyleSheet`, представляющие сами таблицы стилей. Свойство `document.styleSheets` возвращает доступный только для чтения объект, подобный массиву, содержащий объекты `CSSStyleSheet`, представляющие таблицы стилей документа. Если в элементе `<style>` или `<link>`, определяющем или ссылающемся на таблицу стилей, определить атрибут `title`, этот объект будет доступен как свойство объекта `CSSStyleSheet` с именем, указанным в атрибуте `title`.

Следующие разделы описывают, какие операции могут выполняться с этими элементами `<style>` и `<link>` и объектами таблиц стилей.

16.6.1. Включение и выключение таблиц стилей

Простейший прием работы с таблицами стилей является к тому же самым переносимым и надежным. Элементы `<style>` и `<link>` и объекты `CSSStyleSheet` определяют свойство `disabled`, доступное сценариям на языке JavaScript для чтения и записи. Как следует из его имени, если свойство `disabled` принимает значение `true`, таблица стилей оказывается отключенной и будет игнорироваться броузером.

Это наглядно демонстрирует функция `disableStylesheet()`, представленная ниже. Если передать ей число, она будет интерпретировать его как индекс в массиве `document.styleSheets`. Если передать ей строку, она будет интерпретировать ее как селектор CSS, передаст ее методу `document.querySelectorAll()` (раздел 15.2.5) и установит в значение `true` свойство `disabled` всех полученных элементов:

```
function disableStylesheet(ss) {
    if (typeof ss === "number")
        document.styleSheets[ss].disabled = true;
    else {
        var sheets = document.querySelectorAll(ss);
        for(var i = 0; i < sheets.length; i++)
            sheets[i].disabled = true;
    }
}
```

16.6.2. Получение, вставка и удаление правил из таблиц стилей

В дополнение к возможности включения и отключения таблиц стилей объект `CSSStyleSheet` также определяет API для получения, вставки и удаления правил стиля из таблиц стилей. IE8 и более ранние версии реализуют несколько иной API, отличный от стандартного, реализуемого другими броузерами.

Как правило, непосредственное манипулирование таблицами стилей редко бывает полезным. Вместо того чтобы добавлять новые правила в таблицы стилей, обычно лучше оставить их статичными и работать со свойством `className` элемента. В то же время, если необходимо предоставить пользователю возможность полного управления таблицами стилей веб-страницы, может потребоваться организовать динамическое манипулирование таблицами.

Объекты `CSSStyleSheet` хранятся в массиве `document.styleSheets[]`. Объект `CSSStyleSheet` имеет свойство `cssRules[]`, хранящее массив правил стилей:

```
var firstRule = document.styleSheets[0].cssRules[0];
```

В IE это свойство носит имя `rules`, а не `cssRules`.

Элементами массива `cssRules[]` или `rules[]` являются объекты `CSSRule`. В стандартном API объект `CSSRule` может представлять CSS-правила любого типа, включая @-правила, такие как директивы `@import` и `@page`. Однако в IE массив `rules[]` может содержать только фактические правила таблицы стилей.

Объект `CSSRule` имеет два свойства, которые могут использоваться переносимым способом. (В стандартном API правила, не относящиеся к правилам стилей, не имеют этих свойств, и потому, возможно, вам потребуется пропускать их при обходе таблицы стилей.) Свойство `selectorText` – это CSS-селектор для данного правила, а свойство `style` – это ссылка на доступный для записи объект `CSSStyleDeclaration`, который описывает стили, связанные с этим селектором. Напомню, что `CSSStyleDeclaration` – это тот же самый тип, который используется для представления встроенных и вычисленных стилей. Объект CSSStyleDeclaration может применяться для чтения существующих или создания новых стилей в правилах. Нередко при обходе таблицы стилей интерес представляет сам текст правила, а не разобранное его представление. В этом случае можно использовать свойство cssText объекта CSSStyleDeclaration, в котором содержатся правила в текстовом представлении.

В дополнение к возможности получения и изменения существующих правил таблиц стилей имеется возможность добавлять правила в таблицу стилей и удалять их из таблицы. Стандартный прикладной интерфейс определяет методы insertRule() и deleteRule(), позволяющие добавлять и удалять правила:

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

Броузер IE не поддерживает методы insertRule() и deleteRule(), но определяет практически эквивалентные им функции addRule() и removeRule(). Единственное существенное отличие (помимо имен функций) состоит в том, что addRule() ожидает получить селектор и стиль в текстовом виде в двух отдельных аргументах.

Следующий пример реализует обход правил в таблице стилей и демонстрирует API, внося несколько сомнительных изменений в таблицу:

```
var ss = document.styleSheets[0];            // Извлечь первую таблицу стилей
var rules = ss.cssRules?ss.cssRules:ss.rules; // Извлечь правила

for(var i = 0; i < rules.length; i++) {      // Цикл по этим правилам
    var rule = rules[i];
    if (!rule.selectorText) continue;  // Пропустить @import и др. директивы

    var selector = rule.selectorText;  // Селектор
    var ruleText = rule.style.cssText; // Стили в текстовом виде

    // Если правило применяется к элементам h1, применить его к элементам h2
    // Учтите, что этот прием сработает, только если селектор
    // в точности будет иметь вид "h1"
    if (selector == "h1") {
        if (ss.insertRule) ss.insertRule("h2 {" +ruleText+ "}", rules.length);
        else if (ss.addRule) ss.addRule("h2", ruleText, rules.length);
    }

    // Если правило устанавливает свойство text-decoration, удалить его.
    if (rule.style.textDecoration) {
        if (ss.deleteRule) ss.deleteRule(i);
        else if (ss.removeRule) ss.removeRule(i);
        i--; // Скорректировать переменную цикла, поскольку прежнее правило с
             //индексом i+1 теперь стало правилом с индексом i
    }
}
```

16.6.3. Создание новых таблиц стилей

Наконец, имеется возможность создавать совершенно новые таблицы стилей и добавлять их в документ. В большинстве броузеров эта операция выполняется с помощью стандартных приемов, реализованных в модели DOM: создается новый элемент <style> и вставляется в документ в раздел <head>, затем с помощью свойства innerHTML добавляется содержимое таблицы стилей. Однако в IE8 и в более ранних версиях новый объект CSSStyleSheet необходимо создавать с помощью нестандартного метода document.createStyleSheet(), а текст таблицы стилей добавлять с помощью свойства cssText. Пример 16.6 демонстрирует создание новых таблиц.

Пример 16.6. Создание новой таблицы стилей

```
// Добавляет таблицу стилей в документ и заполняет ее указанными стилями.
// Аргумент styles может быть строкой или объектом. Если это строка,
// она интерпретируется как текст таблицы стилей. Если это объект, то каждое
// его свойство должно определять правило стиля, добавляемое в таблицу.
// Именами свойств являются селекторы, а их значениями - соответствующие стили
function addStyles(styles) {
    // Сначала необходимо создать новую таблицу стилей
    var styleElt, styleSheet;
    if (document.createStyleSheet) { //Если определен IE API, использовать его
        styleSheet = document.createStyleSheet();
    }
    else {
        var head = document.getElementsByTagName("head")[0]
        styleElt = document.createElement("style"); // Новый элемент <style>
        head.appendChild(styleElt);                 // Вставить в <head>
        // Теперь новая таблица находится в конце массива
        styleSheet = document.styleSheets[document.styleSheets.length-1]
    }

    // Вставить стили в таблицу
    if (typeof styles === "string") {
        // Аргумент содержит текстовое определение таблицы стилей
        if (styleElt) styleElt.innerHTML = styles;
        else styleSheet.cssText = styles;           // IE API
    }
    else {
        // Аргумент - объект с правилами для вставки
        var i = 0;
        for(selector in styles) {
            if (styleSheet.insertRule) {
                var rule = selector + " {" + styles[selector] + "}";
                styleSheet.insertRule(rule, i++);
            }
            else {
                styleSheet.addRule(selector, styles[selector], i++);
            }
        }
    }
}
```

17

Обработка событий

Клиентские программы на языке JavaScript основаны на модели программирования, когда выполнение программы управляется событиями (представленной в разделе 13.3.2). При таком стиле программирования веб-броузер генерирует событие, когда с документом или некоторым его элементом что-то происходит. Например, веб-броузер генерирует событие, когда завершает загрузку документа, когда пользователь наводит указатель мыши на гиперссылку или нажимает клавишу на клавиатуре. Если JavaScript-приложение интересует определенный тип события для определенного элемента документа, оно может зарегистрировать одну или более функций, которая будет вызываться при возникновении этого события. Имейте в виду, что это не является уникальной особенностью веб-программирования: все приложения с графическим интерфейсом пользователя действуют именно таким образом – они постоянно ожидают, пока что-то произойдет (т. е. ждут появления событий), и откликаются на происходящее.

Обратите внимание, что слово *событие* не является техническим термином, требующим строгого определения. События – это просто некоторые происшествия, о которых броузер извещает программу. События не являются объектами языка JavaScript и никак не описываются в программном коде программы. Однако существует множество объектов, имеющих отношение к событиям, использующихся в программном коде, и эти объекты требуют дополнительного технического описания. Поэтому мы начнем эту главу с некоторых важных определений.

Тип события – это строка, определяющая тип происшествия. Тип «mousemove», например, означает, что пользователь переместил указатель мыши. Тип «keydown» означает, что была нажата клавиша на клавиатуре. А тип «load» означает, что завершилась загрузка документа (или какого-то другого ресурса) из сети. Поскольку тип события – это просто строка, его иногда называют *именем события*. И действительно, мы будем использовать эти имена для идентификации типов событий, о которых будет идти речь. Современные веб-броузеры поддерживают множество типов событий, краткий обзор которых приводится в разделе 17.1.

Цель события – это объект, в котором возникло событие или с которым это событие связано. Когда говорят о событии, обычно упоминают тип и цель события.

Например: событие «load» объекта `Window` или событие «click» элемента `<button>`. Самыми типичными целями событий в клиентских приложениях на языке JavaScript являются объекты `Window`, `Document` и `Element`, но некоторые типы событий могут происходить и в других типах объектов. Например, в главе 18 мы познакомимся с событием «readystatechange», которое возбуждается объектом `XMLHttpRequest`.

Обработчик события, или *приемник события*, – это функция, которая обрабатывает, или откликается на событие.[1] Приложения должны зарегистрировать свои функции обработчиков событий в веб-броузере, указав тип события и цель. Когда в указанном целевом объекте возникнет событие указанного типа, броузер вызовет обработчик. Когда обработчики событий вызываются для какого-то объекта, мы иногда говорим, что броузер «возбудил», «сгенерировал» или «доставил» событие. Существует несколько способов регистрации обработчиков событий, описание которых приводятся в разделах 17.2 и 17.3.

Объект события – это объект, связанный с определенным событием и содержащий информацию об этом событии. Объекты событий передаются функции обработчика события в виде аргумента (кроме IE8 и более ранних версий, где объект события доступен только в виде глобальной переменной `event`). Все объекты событий имеют свойство `type`, определяющее тип события, и свойство `target`, определяющее цель события. (В IE8 в более ранних версиях вместо свойства `target` следует использовать свойство `srcElement`.) Для каждого типа события в связанном объекте события определяется набор свойств. Например, объект, связанный с событиями от мыши, включает координаты указателя мыши, а объект, связанный с событиями от клавиатуры, содержит информацию о нажатой клавише и о нажатых клавишах-модификаторах. Для многих типов событий определяются только стандартные свойства, такие как `type` и `target`, и не передается никакой дополнительной полезной информации. Для таких типов событий важно само наличие происшествия события, и никакая другая информация не имеет значения. В этой главе нет отдельного раздела, посвященного объекту `Event`. Вместо этого здесь описываются свойства объектов событий для событий определенных типов. Дополнительные сведения об объектах событий вы найдете в справочной статье `Event` в четвертой части книги.[2]

Распространение события – это процесс, в ходе которого броузер решает, в каких объектах следует вызвать обработчики событий. В случае событий, предназначенных для единственного объекта (таких как событие «load» объекта `Window`), надобность в их распространении отсутствует. Однако, когда некоторое событие возникает в элементе документа, оно распространяется, или «всплывает», вверх по дереву документа. Если пользователь щелкнет мышью на гиперссылке, событие «mousemove» сначала будет возбуждено в элементе `<a>`, определяющем эту ссылку. Затем оно будет доставлено вмещающим элементам: возможно, элементу `<p>`, элементу `<div>` и самому объекту `Document`. Иногда удобнее бывает зарегистрировать единственный обработчик события в объекте `Document` или в другом контейнерном элементе, чем выполнять регистрацию во всех интересующих нас элементах. Обработчик события может прервать дальнейшее распространение события, чтобы оно прекратило всплытие и не привело к вызову обработчиков вмещающих элементов. Делается это вызовом метода или установкой свойства объекта события. Детально распространение событий рассматривается в разделе 17.3.6.

---

[1] Некоторые источники, включая спецификацию HTML5, различают обработчики событий и приемники событий, в соответствии со способом, которым они были зарегистрированы. В этой книге мы будем использовать эти два термина как синонимы.

[2] Стандарты определяют целую иерархию объектов событий для событий различных типов. Интерфейс `Event` описывает «простые» события, которые не сопровождаются дополнительной информацией. Например, подкласс `MouseEvent` описывает дополнительные поля, доступные в объектах событий, передаваемых при возбуждении события от мыши, а подкласс `KeyEvent` описывает поля, которые можно использовать при обработке событий от клавиатуры. Описания всех этих классов в данной книге помещены в общую справочную статью `Event`.

Еще одна форма распространения событий, которая называется *перехватом события*, позволяет специально зарегистрированным обработчикам или контейнерным элементам «перехватывать» события до того, как они достигнут фактической цели. Перехват событий не поддерживается в IE8 и в более ранних версиях и поэтому редко используется на практике. Однако возможность перехватывать события от мыши совершенно необходима при обработке событий буксировки объектов мышью; как это делается, будет показано в примере 17.2.

Для некоторых событий предусматриваются связанные с ними *действия по умолчанию*. Например, для события «click», возникающего в гиперссылке, по умолчанию предусматривается операция перехода по ссылке и загрузки новой страницы. Обработчики могут предотвратить выполнение действий по умолчанию, вернув соответствующее значение, вызвав метод или установив свойство объекта события. Это иногда называют «отменой» события; подробнее эта тема рассматривается в разделе 17.3.7.

Дав определения некоторым терминам, можно перейти к изучению вопросов, связанных с событиями и их обработкой. В первом разделе, следующем ниже, приводится обзор множества типов событий, поддерживаемых веб-броузерами. Вы не найдете здесь подробное описание какого-то одного типа событий, но этот раздел даст вам представление о том, какие типы событий доступны для использования в веб-приложениях. В данном разделе есть ссылки на другие части книги, где демонстрируются некоторые события в действии.

После представления типов событий в следующих двух разделах описывается, как регистрировать обработчики событий и как броузер вызывает эти обработчики событий. Из-за особенностей развития модели событий в JavaScript и из-за отсутствия поддержки стандартной модели в IE8 и в более ранних версиях обе эти темы являются более сложными, чем можно было бы ожидать.

Завершится эта глава серией примеров, демонстрирующих, как правильно обрабатывать некоторые типы событий. В этих разделах рассматриваются:

- События загрузки и готовности документа
- События от мыши
- Событие от колесика мыши
- События буксировки мышью
- События от клавиатуры
- События ввода текста

17.1. Типы событий

На заре развития Всемирной паутины веб-разработчикам приходилось иметь дело лишь с небольшим количеством событий: «load», «click», «mouseover» и другими. Эти довольно старые типы событий хорошо поддерживаются всеми броузерами и рассматриваются в разделе 17.1.1. По мере развития веб-платформы в нее были включены более мощные прикладные интерфейсы, а количество событий существенно увеличилось. Не существует стандарта, который определял бы полный набор событий, и к моменту написания этих строк количество поддерживаемых событий продолжает быстро увеличиваться. Эти новые события определяются в следующих трех источниках:

- Спецификация «DOM Level 3 Events», которая после долгих лет застоя стала активно разрабатываться под эгидой консорциума W3C. События DOM рассматриваются в разделе 17.1.2.
- Множество новых прикладных интерфейсов в спецификации HTML5 (и в связанных с ней дополнительных спецификациях) определяют новые события, используемые, например, для управления историей посещений, механизмом drag-and-drop (перетащил и бросил), обмена сообщениями между документами и проигрывания аудио- и видеороликов. Обзор этих событий приводится в разделе 17.1.3.
- С появлением мобильных устройств с сенсорными экранами, поддерживающих JavaScript, таких как iPhone, возникла необходимость в определении новых типов событий касаний и жестов. Некоторые примеры, специфические для устройств, выпускаемых компанией Apple, приводятся в разделе 17.1.4.

Обратите внимание, что многие из этих новых типов событий пока еще не получили широкой поддержки и определяются стандартами, которые еще находятся на стадии проектирования. В последующих разделах дается краткий обзор этих событий, но подробное их описание отсутствует. Оставшаяся часть главы полностью охватывает модель обработки событий и включает множество примеров работы с поддерживаемыми событиями. Если вы поймете, как работать с событиями в целом, вы легко сможете реализовать обработку этих новых типов событий, как только будут определены и реализованы соответствующие прикладные интерфейсы.

17.1.1. Старые типы событий

События, которые вам чаще всего придется использовать в своих веб-приложениях, обычно будут относиться к категории давно существующих и поддерживаемых всеми броузерами: это события для работы с мышью, с клавиатурой, с HTML-формами и с объектом `Window`. В следующих разделах описывается множество важных особенностей событий этих типов.

17.1.1.1. События форм

Формы и гиперссылки стали первыми элементами веб-страниц, возможность управления которыми была реализована в начале развития Всемирной паутины и JavaScript. Это означает, что события форм являются наиболее устойчивыми и хорошо поддерживаемыми из всех типов событий. Элементы `<form>` возбуждают события «submit», при отправке формы, и «reset», перед сбросом формы в исходное состояние. Элементы форм, внешним видом напоминающие кнопки (включая радиокнопки и флажки), возбуждают события «click», когда пользователь взаимодействуют с ними. Элементы форм, позволяющие вводить некоторую информацию, обычно возбуждают события «change», когда пользователь изменяет их состояние, вводя текст, выбирая элемент списка или отмечая флажок. Для текстовых элементов ввода событие «change» не возбуждается, пока пользователь не завершит взаимодействие с ними и не передаст фокус ввода другому элементу. Элементы форм откликаются на изменение фокуса ввода, возбуждая события «focus» и «blur» при получении и утере фокуса ввода.

Все эти события, связанные с формами, подробно рассматривались в разделе 15.9.3. Тем не менее здесь следует сделать несколько важных дополнений.

Категории событий

События могут быть сгруппированы в некоторые обобщенные категории, знание которых поможет вам понять длинный перечень событий, следующий далее:

Аппаратно-зависимые события ввода

События из этой категории непосредственно связаны с конкретными устройствами ввода, такими как мышь или клавиатура. В эту категорию входят такие старые события, как «mousedown», «mousemove», «mouseup», «keydown», «keypress» и «keyup», а также новые события, имеющие отношение к сенсорным устройствам, такие как «touchmove» и «gesturechange».

Аппаратно-независимые события ввода

Эти события ввода не связаны непосредственно с каким-то определенным устройством. Например, событие «click» указывает на то, что была активирована ссылка или кнопка (или другой элемент документа). Это событие часто порождается щелчком мыши, но его причиной также может быть нажатие клавиши на клавиатуре или (для сенсорных устройств) некоторое перемещение по экрану. Событие «textinput» (которое пока реализовано не во всех броузерах) является аппаратно-независимой альтернативой событию «keypress» и поддерживает не только ввод с клавиатуры, но и такие альтернативы, как вставка из буфера обмена и ввод рукописного текста.

События пользовательского интерфейса

События ПИ – это высокоуровневые события, которые часто возникают в элементах HTML-форм, составляющих пользовательский интерфейс веб-приложения. В эту категорию входит событие «focus» (возникающее, когда текстовое поле получает фокус ввода), событие «change» (возникающее, когда пользователь изменяет значение, отображаемое элементом формы) и событие «submit» (возникающее, когда пользователь щелкает на кнопке отправки формы).

События изменения состояния

Некоторые события не связаны непосредственно с деятельностью пользователя, но имеют отношение к выполнению сетевых операций броузером и указывают на переход к другому этапу операции или на изменение состояния. Событие «load», которое возбуждается в объекте `Window` по окончании загрузки документа, является, пожалуй, наиболее часто используемым типом событий из этой категории. Еще одним представителем из этой категории является событие «DOMContentLoaded» (обсуждалось в разделе 13.3.4). Механизм управления историей посещений, определяемый стандартом HTML5 (раздел 22.2), возбуждает событие «popstate» в ответ на нажатие клавиши Back (Назад) броузера. Прикладной интерфейс автономных веб-приложений, описываемый стандартом HTML5 (раздел 20.4), включает события «online» и «offline». В главе 18 демонстрируется, как пользоваться событием «readystatechange», сообщающем о получении данных с сервера. Аналогично новый API чтения локальных файлов, выбранных пользователем (раздел 22.6.5), использует события, такие как «loadstart», «progress» и «loadend», для отправки асинхронных извещений о ходе выполнения операций ввода-вывода.

Прикладные события

Некоторые прикладные интерфейсы, определяемые стандартом HTML5 и связанными с ним спецификациями, включают собственные типы событий. Интерфейс drag-and-drop (раздел 17.7) определет такие события, как «dragstart», «dragenter», «dragover» и «drop». Приложения, обеспечивающие поддержку буксировки элементов мышью, должны реализовать обработку некоторых из этих событий. Элементы `<video>` и `<audio>` (раздел 21.2), определяемые стандартом HTML5, добавляют длинный список связанных с ними типов событий, таких как «waiting», «playing», «seeking», «volumechange» и т. д. Эти события обычно представляют интерес только для веб-приложений, определяющих собственные элементы управления проигрыванием аудио- и видеороликов.

Обработчики ошибок и событий от таймеров

Обработчики ошибок и событий от таймеров (были описаны в главе 14) являются частью асинхронной модели программирования в клиентском JavaScript и похожи на обработчики обычных событий. Несмотря на то что обработчики ошибок и событий от таймеров не рассматриваются в этой главе, их очень удобно воспринимать как обработчики обычных событий, и, возможно, вам будет интересно прочитать разделы 14.1 и 14.6 еще раз, с позиций, предлагаемых этой главой.

Для событий «submit» и «reset» предусматриваются действия по умолчанию, выполнение которых можно отменить в обработчиках событий, как и в случае некоторых событий «click». Все события форм всплывают, кроме событий «focus» и «blur». IE определяет события «focusin» и «focusout», которые являются всплывающими альтернативами событий «focus» и «blur». Библиотека jQuery (глава 19) имитирует события «focusin» и «focusout» в броузерах, не поддерживающих их.

Кроме того, эти события были стандартизованы спецификацией «DOM Level 3 Events».

Наконец, имейте в виду, что все броузеры, кроме IE, возбуждают событие «input» в элементах `<textarea>` и в других текстовых элементах ввода, когда пользователь вводит текст (посредством клавиатуры или вставкой из буфера обмена) в элемент. В отличие от события «change», данное событие «input» возбуждается при каждой вставке. К сожалению, объект события, соответствующий событию «input», не позволяет узнать, какой текст был введен. (Более полезной альтернативой этому событию является новое событие «textinput», описываемое ниже.)

17.1.1.2. События объекта Window

События объекта `Window` представляют происшествия, имеющие отношение к самому окну броузера, а не к определенному содержимому документа, отображаемому в окне. (Однако некоторые из этих событий имеют имена, совпадающие с именами событий, возбуждаемых для элементов документа.)

Самым важным из этих событий является событие «load»: оно возбуждается сразу после того, как будут загружены и отображены документ и все внешние ресурсы (такие как изображения). Событие «load» обсуждалось на протяжении главы 13. Альтернативами событию «load» являются события «DOMContentLoaded» и «readystatechange»: они возбуждаются сразу же, как только документ и его элементы будут готовы к выполнению операций, но до того, как полностью будут загружены внешние ресурсы. Примеры использования этих событий, имеющих отношение к документу, приводятся в разделе 17.4.

Событие «unload» является противоположностью событию «load»: оно возбуждается, когда пользователь покидает документ. Обработчик события «unload» можно использовать, чтобы сохранить информацию о состоянии, но в нем нельзя отменить переход на другую страницу. Событие «beforeunload» похоже на событие «unload», но оно дает возможность узнать у пользователя, действительно ли он желает покинуть вашу веб-страницу. Если обработчик события «beforeunload» вернет строку, эта строка будет выведена в диалоге подтверждения перед тем, как будет загружена новая страница; этот диалог даст пользователю возможность отменить переход и остаться на текущей странице.

Свойство `onerror` объекта `Window` является своего рода обработчиком событий, который вызывается в случае появления ошибок в программном коде на языке JavaScript. Однако это не настоящий обработчик событий, потому что он вызывается совсем с другим набором аргументов. Подробности смотрите в разделе 14.6.

Имеется также возможность регистрировать обработчики событий «load» и «error» для отдельных элементов документа, таких как `<img>`. Эти события возбуждаются, когда внешний ресурс (например, изображение) будет полностью загружен или когда возникнет ошибка, препятствующая его загрузке. Некоторые броузеры поддерживают также событие «abort» (стандартизованное спецификацией HTML5), которое возбуждается, когда загрузка изображения (или другой ресурс, загружаемый из сети) прерывается из-за того, что пользователь остановил процесс загрузки.

События «focus» и «blur», описанные выше вместе с другими событиями элементов форм, также поддерживаются объектом `Window`: они возбуждаются, когда текущее окно броузера получает или теряет фокус ввода.

Наконец, события «resize» и «scroll» возбуждаются в объекте `Window`, когда выполняется изменение размеров или прокрутка окна броузера. События «scroll» могут также возбуждать все прокручиваемые элементы документа, например те, что имеют CSS-свойство `overflow` (раздел 16.2.6). Объект события, передаваемый обработчикам событий «resize» и «scroll», – это самый обычный объект `Event`, не имеющий свойств, которые позволяли бы узнать новый размер окна или величину прокрутки. Чтобы узнать новый размер окна или позиции полос прокрутки, следует использовать приемы, продемонстрированные в разделе 15.8.

17.1.1.3. События мыши

События от мыши возбуждаются, когда пользователь перемещает указатель мыши или выполняет щелчок. Эти события генерируются в наиболее глубоко вложенных элементах, над которыми находится указатель мыши, но они всплывают вверх по дереву документа. Объект события, передаваемый обработчикам событий от мыши, имеет свойства, позволяющие узнать координаты указателя, состояние кнопок мыши, а также состояние клавиш-модификаторов на момент возникновения события. Свойства `clientX` и `clientY` определяют положение указателя мыши в системе координат окна. Свойства `button` и `which` позволяют узнать, какая кнопка была нажата. (Обязательно ознакомьтесь со справочной статьей `Event` в четвертой части книги, чтобы знать, как организовать работу с этими свойствами переносимым способом.) Свойства `altKey`, `ctrlKey`, `metaKey` и `shiftKey` получают значение `true`, если в момент возникновения события удерживалась нажатой соответствующая клавиша-модификатор. А свойство `detail` для события «click» указывает, был ли выполнен одинарный, двойной или тройной щелчок.

Событие «mousemove» генерируется всякий раз, когда пользователь перемещает указатель мыши. Это событие возбуждается очень часто, поэтому его обработчики не должны выполнять тяжелые вычисления. События «mousedown» и «mouseup» генерируются, когда пользователь нажимает и отпускает кнопку мыши. Зарегистрировав обработчик события «mousedown», который регистрирует обработчик события «mousemove», можно организовать определение и обработку ситуаций буксировки элементов мышью. При этом необходимо обеспечить возможность перехватывать события от мыши, чтобы продолжать получать события «mousemove», даже когда указатель мыши выходит за пределы элемента, где была начата буксировка. В разделе 17.5 приводится пример реализации буксировки элементов мышью.

После последовательности событий «mousedown» и «mouseup» броузер генерирует событие «click». Событие «click» было описано выше, как аппаратно-независимое событие форм, но на самом деле оно может генерироваться для любого элемента документа, а не только для элементов форм, и вместе с ним обработчику передается объект события со всеми дополнительными полями, описанными выше. Если вы дважды щелкнете мышью на строке (в течение достаточно короткого промежутка времени), второй щелчок сгенерирует событий «dblclick». Когда щелчок выполняется правой кнопкой мыши, броузеры часто выводят контекстное меню. Вообще, прежде чем вывести контекстное меню, они возбуждают событие «contextmenu», и, если отменить это событие в обработчике, можно предотвратить появление меню. Кроме того, это наиболее простой способ организовать обработку щелчка правой кнопкой мыши.

Когда пользователь перемещает указатель мыши так, что он оказывается над другим элементом, броузер возбуждает событие «mouseover» для этого элемента.

Когда указатель мыши покидает границы элемента, броузер генерирует для него событие «mouseout». Объект события для этих событий будет иметь свойство `relatedTarget`, определяющее другой элемент, вовлеченный в переход. (Эквивалент свойства `relatedTarget` в IE вы найдете в справочной статье `Event`.) События «mouseover» и «mouseout» всплывают, как и все остальные описанные здесь события от мыши. Иногда это оказывается неудобно, потому что в обработчике события «mouseout» приходится проверять, действительно ли указатель мыши покинул данный элемент или он просто переместился из одного дочернего элемента в другой. По этой причине в IE поддерживается невсплывающие версии этих событий, известные как «mouseenter» и «mouseleave». Библиотека jQuery эмулирует поддержку этих событий в броузерах, отличных от IE (глава 19), а кроме того, эти события стандартизованы в спецификации «DOM Level 3 Events».

Когда пользователь вращает колесико мыши, броузеры генерируют событие «mousewheel» (или в Firefox событие «DOMMouseScroll»). Объект события, передаваемый вместе с этими событиями, включает свойства, позволяющие узнать, на какое расстояние и в каком направлении было повернуто колесико. Спецификация «DOM Level 3 Events» стандартизует более универсальное многомерное событие от колесика, которое, если будет реализовано, заменит оба события, «mousewheel» и «DOMMouseScroll». Пример обработки события «mousewheel» приводится в разделе 17.6.

17.1.1.4. События клавиатуры

Когда веб-броузер получает фокус ввода, он начинает генерировать события всякий раз, когда пользователь нажимает и отпускает клавиши на клавиатуре. Нажатия горячих комбинаций, имеющих значение для операционной системы или самого броузера, часто «съедаются» операционной системой или броузером и не передаются обработчикам событий на JavaScript. События от клавиатуры генерируются в любом элементе документа, обладающем фокусом ввода, и всплывают вверх до объектов документа и окна. Если ни один элемент не обладает фокусом ввода, события возбуждаются непосредственно в объекте документа. Обработчикам событий от клавиатуры передается объект события, имеющий свойство `keyCode`, позволяющее узнать, какая клавиша была нажата или отпущена. В дополнение к свойству `keyCode` объект события от клавиатуры также имеет свойства `altKey`, `ctrlKey`, `metaKey` и `shiftKey`, описывающие состояние клавиш-модификаторов.

События «keydown» и «keyup» являются низкоуровневыми событиями от клавиатуры: они генерируются, когда производится нажатие или отпускание клавиши (даже если это клавиша-модификатор). Когда событие «keydown» генерируется нажатием клавиши, соответствующей печатаемому символу, после события «keydown», но перед событием «keyup» дополнительно генерируется событие «keypress». (В случае если клавиша удерживается в нажатом состоянии настолько долго, что начинается автоповтор символа, перед событием «keyup» будет сгенерировано множество событий «keypress».) Событие «keypress» является высокоуровневым событием ввода текста и соответствующий ему объект события содержит информацию о введенном символе, а не о нажатой клавише.

События «keydown», «keyup» и «keypress» поддерживаются всеми броузерами, однако существуют некоторые проблемы совместимости из-за того, что не были стандартизованы значения свойства `keyCode` объекта события. Спецификация «DOM Level 3 Events», описываемая ниже, пытается решить эти проблемы совместимости, но эти решения пока не реализованы. Пример обработки событий «keydown» приводится в разделе 17.9, а в разделе 17.8 приводится пример обработки событий «keypress».

17.1.2. События модели DOM

Спецификация «DOM Level 3 Events» разрабатывалась консорциумом W3C около десяти лет. К моменту написания этих строк она была подвергнута существенному пересмотру с целью привести в соответствие с текущими реалиями и наконец достигла стадии стандартизации «последней версии рабочего проекта». Она стандартизует многие из старых событий, описанных выше, и добавляет несколько новых событий, описываемых здесь. Эти новые типы событий пока не получили широкой поддержки, но производители броузеров предполагают реализовать их к моменту окончательного утверждения стандарта.

Как отмечалось выше, спецификация «DOM Level 3 Events» стандартизует события «focusin» и «focusout» как всплывающие альтернативы событий «focus» и «blur», а также события «mouseenter» и «mouseleave» – как невсплывающие альтернативы событий «mouseover» и «mouseout». Кроме того, эта версия стандарта не рекомендует использовать некоторые типы событий, которые были определены спецификацией «DOM Level 2 Events», но никогда не были реализованы. Броузеры по-прежнему обеспечивают поддержку событий, таких как «DOMActivate», «DOMFocusIn» и «DOMNodeInserted», но теперь это необязательно, и потому данные события не описываются в этой книге.[1]

Из новшеств, появившихся в спецификации «DOM Level 3 Events», можно назвать стандартизацию поддержки двунаправленных колесиков мыши через событие «wheel» и улучшенную поддержку событий ввода текста через событие «textinput» и новый объект `KeyboardEvent`, который передается обработчикам событий «keydown», «keyup» и «keypress».

Согласно этой спецификации обработчику события «wheel» должен передаваться объект события, содержащий все свойства, обычные для объектов событий от мыши, а также свойства `deltaX`, `deltaY` и `deltaZ`, позволяющие узнать величину прокрутки вокруг трех разных осей колесика мыши. (В большинстве мышей колесико вращается в одном или двух измерениях, и поэтому свойство `deltaZ` пока остается неиспользуемым.) Подробнее о событиях «mousewheel» рассказывается в разделе 17.6.

Стандарт «DOM Level 3 Events» определяет событие «keypress», описанное выше, но не рекомендует использовать его и отдает предпочтение новому событию с именем «textinput». Вместо сложного в использовании числового значения в свойстве `keyCode`, объект события, передаваемый обработчикам события «textinput», имеет свойство `data`, содержащее введенную строку текста. Событие «textinput» не является в полной мере событием от клавиатуры: оно возбуждается при выполнении любой операции ввода текста, которая может быть выполнена с помощью клавиатуры, копированием из буфера обмена, операцией буксировки (drag-and-drop) и т. д. Спецификация определяет свойство `inputMethod` объекта события и множество констант, представляющих различные способы ввода текста (с клавиатуры, копированием из буфера обмена или буксировкой мышью, путем распознавания рукописного текста или голоса и т. д.). К моменту написания этих строк броузеры Safari и Chrome поддерживали версию этого события с именем «textInput». Соответствующий ему объект события включает свойство `data`, но в нем отсутствует свойство `inputMethod`. Пример использования события «textInput» приводится в разделе 17.8.

---

[1] Единственным часто используемым событием, в имени которого присутствует приставка «DOM», является событие «DOMContentLoaded». Это событие было введено компанией Mozilla и никогда не являлось частью стандарта «DOM Events».

Новый стандарт DOM также упрощает события «keydown», «keyup» и «keypress», добавляя новые свойства `key` и `char` в объект события. Оба эти свойства содержат строковые значения. Для клавиш, генерирующих печатаемые символы, свойства `key` и `char` будут хранить один и тот же сгенерированный текст. Для управляющих клавиш свойство `key` будет хранить строку вида «Enter», «Delete» или «Left», идентифицирующую клавишу. А свойство `char` будет хранить либо значение `null`, либо, для таких управляющих клавиш, как Tab, – имеющих соответствующий управляющий символ, – строку, сгенерированную клавишей. На момент написания этих строк ни один броузер не поддерживал эти свойства `key` и `char`, но пример 17.8 будет использовать свойство `key`, когда оно будет реализовано.

17.1.3. События HTML5

Стандарт HTML5 и связанные с ним стандарты определяют основу новых API для веб-приложений (глава 22). Многие из этих API определяют события. В этом разделе перечисляются и коротко описываются эти события HTML5 и веб-приложений. Некоторые из этих событий уже готовы к использованию и более подробно описываются в разных главах книги. Другие пока реализованы не во всех броузерах и не описываются подробно.

Одной из широко рекламируемых особенностей HTML является возможность включения элементов `<audio>` и `<video>` для проигрывания аудио- и видеороликов. Эти элементы имеют длинный перечень генерируемых ими событий, позволяющих отправлять извещения о сетевых событиях, о состоянии механизма буферизации данных и механизма воспроизведения:

`canplay`
`canplaythrough`
`durationchange`
`emptied`
`ended`
`loadeddata`
`loadedmetadata`
`loadstart`
`pause`
`play`
`playing`
`progress`
`ratechange`
`seeked`
`seeking`
`stalled`
`suspend`
`timeupdate`
`volumechange`
`waiting`

Эти события, имеющие отношение к медиапроигрывателям, передаются в виде простого объекта события, не имеющего специальных свойств. Однако свойство `target` идентифицирует элемент `<audio>` или `<video>`, и этот элемент имеет множество специфических свойств и методов. Более подробно об этих элементах, их свойствах и событиях рассказывается в разделе 21.2.

Интерфейс механизма буксировки (drag-and-drop), определяемый стандартом HTML5, позволяет приложениям на языке JavaScript участвовать в операциях буксировки объектов мышью, опираясь на механизмы, реализованные в операционной системе, и обмениваться данными с обычными приложениями. Этот прикладной интерфейс определяет следующие семь типов событий:

`dragstart`
`dragenter`
`drop`
`drag`
`dragover`
`dragend`
`dragleave`

Эти события буксировки сопровождаются объектами событий, подобными тем, что передаются вместе с событиями от мыши. Отличаются они единственным дополнительным свойством `dataTransfer`, хранящим объект `DataTransfer` с информацией о передаваемых данных и о форматах, в которых эти данные доступны. Прикладной интерфейс механизма drag-and-drop, определяемый стандартом HTML5, описывается и демонстрируется в разделе 17.7.

Спецификация HTML5 определяет также механизм управления историей посещений (раздел 22.2), что позволяет веб-приложениям взаимодействовать с кнопками броузера Back (Назад) и Forward (Вперед). Этот механизм вводит события с именами «hashchange» и «popstate», которые возникают тогда же, когда и события «load» и «unload», и возбуждаются в объекте `Window`, а не в документе.

В HTML5 определяется множество новых особенностей HTML-форм. В дополнение к событиям ввода, описанным выше, спецификация HTML5 также определяет механизм проверки форм, который привносит событие «invalid», возбуждаемое в элементе формы, не прошедшем проверку. Однако производители броузеров, кроме Opera, не торопятся воплощать новые особенности форм и новые события, поэтому они не рассматриваются в этой книге.

Спецификация HTML5 включает поддержку веб-приложений, способных выполняться без подключения к сети (раздел 20.4), которые могут быть установлены локально в кэше приложений, чтобы их можно было запускать, даже когда броузер работает в автономном режиме (например, когда мобильное устройство находится вне сети). С этой поддержкой связаны два наиболее важных события, «offline» и «online»: они генерируются в объекте `Window`, когда броузер теряет или обретает соединение с сетью. Кроме того, определено несколько дополнительных событий, посредством которых приложение извещается о ходе выполнения загрузки и обновления кэша приложений:

```
cached      checking    downloading   error
noupdate    obsolete    progress      updateready
```

Событие «message» используется множеством новых API веб-приложений для организации асинхронных взаимодействий. Прикладной интерфейс взаимодействий между документами (раздел 22.3) позволяет сценариям в документе с одного сервера обмениваться сообщениями со сценариями в документе с другого сервера. Это дает возможность безопасно обойти ограничения политики общего происхождения (раздел 13.6.2). При передаче каждого сообщения в объекте `Window` документа, принимающего сообщение, генерируется событие «message». Объект события, передаваемый обработчику, включает свойство `data`, хранящее содержимое сообщения, а также свойства `source` и `origin`, идентифицирующие отправителя сообщения. Событие «message» используется также для взаимодействия с фоновыми потоками Web Workers (раздел 22.4) и для сетевых взаимодействий посредством прикладных интерфейсов, определяемых спецификациями «Server-Sent Events» (раздел 18.3) и «WebSockets» (раздел 22.9).

Стандарт HTML5 и связанные с ним спецификации определяют некоторые события, генерируемые в объектах, не являющихся окнами, документами и элементами документов. Версия 2 спецификации «XMLHttpRequest», а также спецификация «File API» определяют множество событий, помогающих следить за ходом выполнения асинхронных операций ввода/вывода. Эти события генерируются в объектах `XMLHttpRequest` или `FileReader`. Каждая операция чтения начинается с события «loadstart», за которым следует последовательность событий «progress» и событие «loadend». Кроме того, каждая операция завершается событием «load», «error» или «abort», генерируемым непосредственно перед заключительным событием «loadend». Подробнее об этих событиях рассказывается в разделах 18.1.4 и 22.6.5.

Наконец, стандарт HTML5 и связанные с ним спецификации определяют еще несколько различных типов событий. Спецификация «Web Storage API» (раздел 20.1) определяет событие «storage» (генерируемое в объекте `Window`), извещающее об изменении хранимых данных. В спецификации HTML5 также определены события «beforeprint» и «afterprint», впервые введенные компанией Microsoft в IE. Как следует из их имен, эти события генерируются в окне `Window` непосредственно до и после того, как документ будет напечатан, и предоставляют возможность добавить или удалить содержимое, такое как дата и время печати документа. (Эти события не должны использоваться для изменения представления документа для печати, потому что для этой цели в CSS уже имеются директивы определения типа носителя.)

17.1.4. События, генерируемые сенсорными экранами и мобильными устройствами

Широкое распространение мощных мобильных устройств, особенно устройств с сенсорными экранами, потребовало создания новых категорий событий. Во многих случаях события от сенсорных экранов отображаются на традиционные типы событий, такие как «click» и «scroll». Но не все виды взаимодействий с пользовательским интерфейсом через сенсорный экран можно имитировать с помощью мыши, и не все прикосновения к такому экрану можно интерпретировать, как события от мыши. В этом разделе кратко описываются жесты и события прикосновений, генерируемые броузером Safari, когда он выполняется на устройствах iPhone и iPad компании Apple, а также рассматривается событие «orientationchange», генерируемое, когда пользователь поворачивает устройство. На момент написания этих строк данные события не были стандартизованы, но консорциум W3C уже приступил к работе над спецификацией «Touch Events Specification», в которой за основу приняты события прикосновения, внедренные компанией Apple. Эти события не описываются в справочном разделе данной книги, но дополнительную информацию вы сможете найти на сайте Apple Developer Center (*http://developer.apple.com/*).

Броузер Safari генерирует события для жестов масштабирования и вращения из двух пальцев. Событие «gesturestart» возбуждается, когда начинается выполнение жеста, а событие «gestureend» по его окончании. Между этими двумя событиями генерируется последовательность событий «gesturechange», позволяющих отслеживать выполнение жеста. Объект события, передаваемый вместе с этими событиями, имеет числовые свойства `scale` и `rotation`. Свойство `scale` определяет отношение текущего и начального расстояний между двумя пальцами. Для жеста, когда пальцы сводятся, свойство `scale` получает значение меньше 1.0, а для жеста, когда пальцы разводятся, свойство `scale` получает значение больше 1.0. Свойство `rotation` определяет угол поворота пальцев с момента события «gesturestart». Значение угла всегда является положительным и измеряется в градусах по часовой стрелке.

События жестов являются высокоуровневыми событиями, извещающими о жесте, уже прошедшем интерпретацию. Реализовать поддержку собственных жестов можно с помощью обработчиков низкоуровневых событий прикосновений. Когда палец касается экрана, генерируется событие «touchstart». Когда палец перемещается, генерируется событие «touchmove». А когда палец отнимается от экрана, генерируется событие «touchend». В отличие от событий мыши, события прикосновений не несут непосредственной информации о координатах прикосновения. Вместо этого в объекте события, который поставляется вместе с событием прикосновения, имеется свойство `changedTouches`. Это свойство хранит объект, подобный массиву, каждый элемент которого описывает позицию прикосновения.

Событие «orientationchanged» генерируется в объекте `Window` устройствами, позволяющими пользователям поворачивать экран для перехода из книжной ориентации в альбомную. Объект, передаваемый вместе с событием «orientationchanged», не очень полезен сам по себе. Однако в мобильной версии броузера Safari объект `Window` имеет свойство `orientation`, определяющее текущую ориентацию в виде числовых значений 0, 90, 180 или -90.

17.2. Регистрация обработчиков событий

Существует два основных способа регистрации обработчиков событий. Первый, появившийся на раннем этапе развития Всемирной паутины, заключается в установке свойства объекта или элемента документа, являющегося целью события. Второй способ, более новый и более универсальный, заключается в передаче обработчика методу объекта или элемента. Дело осложняется тем, что каждый прием имеет две версии. Свойство обработчика события можно установить в программном коде на языке JavaScript или в элементе документа, определив соответствующий атрибут непосредственно в разметке HTML. Регистрация обработчиков вызовом метода может быть выполнена стандартным методом с именем `addEventListener()`, который поддерживается всеми броузерами, кроме IE версии 8 и ниже, и другим методом, с именем `attachEvent()`, поддерживаемым всеми версиями IE до IE9.

17.2.1. Установка свойств обработчиков событий

Самый простой способ зарегистрировать обработчик события заключается в том, чтобы присвоить свойству целевого объекта события желаемую функцию обработчика. По соглашению свойства обработчиков событий имеют имена, состоящие из слова «on», за которым следует имя события: `onclick`, `onchange`, `onload`, `onmouseover` и т. д. Обратите внимание, что эти имена свойств чувствительны к регистру и в них используются только строчные символы, даже когда имя типа события состоит из нескольких слов (например «readystatechange»). Ниже приводятся два примера регистрации обработчиков событий:

```
// Присвоить функцию свойству onload объекта Window.
// Функция - обработчик события: она вызывается, когда документ будет загружен.
window.onload = function() {
    // Отыскать элемент <form>
    var elt = document.getElementById("shipping_address");
    // Зарегистрировать обработчик события, который будет вызываться
    // непосредственно перед отправкой формы.
```

```
        elt.onsubmit = function() { return validate(this); }
    }
```

Такой способ регистрации обработчиков событий поддерживается во всех броузерах для всех часто используемых типов событий. Вообще говоря, все прикладные интерфейсы, получившие широкую поддержку, которые определяют свои события, позволяют регистрировать обработчики установкой свойств обработчиков событий.

Недостаток использования свойств обработчиков событий состоит в том, что они проектировались в предположении, что цели событий будут иметь не более одного обработчика для каждого типа событий. При создании библиотеки для использования в произвольных документах для регистрации обработчиков лучше использовать прием (такой как вызов метода `addEventListener()`), не изменяющий и не затирающий ранее зарегистрированные обработчики.

17.2.2. Установка атрибутов обработчиков событий

Свойства обработчиков событий в элементах документа можно также устанавливать, определяя значения атрибутов в соответствующих HTML-тегах. В этом случае значение атрибута должно быть строкой программного кода на языке JavaScript. Этот программный код должен быть не полным объявлением функции обработчика события, а только ее телом. То есть реализация обработчика события в разметке HTML не должна заключаться в фигурные скобки и предваряться ключевым словом `function`. Например:

```
<button onclick="alert('Спасибо');">Щелкните здесь</button>
```

Если значение HTML-атрибута обработчика события состоит из нескольких JavaScript-инструкций, они должны отделяться точками с запятой либо значение атрибута должно располагаться в нескольких строках.

Некоторые типы событий предназначены для броузера в целом, а не для какого-то конкретного элемента документа. Обработчики таких событий в языке JavaScript регистрируются в объекте `Window`. В разметке HTML они должны помещаться в тег `<body>`, но броузер зарегистрирует их в объекте `Window`. Ниже приводится полный список таких обработчиков событий, определяемых проектом спецификации HTML5:

`onafterprint`
`onbeforeprint`
`onbeforeunload`
`onblur`
`onerror`
`onfocus`
`onhashchange`
`onload`
`onmessage`
`onoffline`
`ononline`
`onpagehide`
`onpageshow`
`onpopstate`
`onredo`
`onresize`
`onstorage`
`onundo`
`onunload`

Когда в качестве значения атрибута обработчика события в разметке HTML указывается строка с программным кодом на языке JavaScript, броузер преобразует эту строку в функцию, которая будет выглядеть примерно так:

```
function(event) {
    with(document) {
        with(this.form || {}) {
            with(this) {
                /* ваш программный код */
            }
            }
        }
    }
```

Если броузер поддерживает стандарт ES5, функция определяется в нестрогом режиме (раздел 5.7.3). Мы еще встретимся с аргументом `event` и инструкциями `with`, когда будем рассматривать вызов обработчиков событий в разделе 17.3.

При разработке клиентских сценариев обычно принято отделять разметку HTML от программного кода на языке JavaScript. Программисты, следующие этому правилу, избегают (или, по крайней мере, стараются избегать) использовать HTML-атрибуты обработчиков событий, чтобы не смешивать программный код на языке JavaScript и разметку HTML.

17.2.3. addEventListener()

В стандартной модели событий, поддерживаемой всеми броузерами, кроме IE версии 8 и ниже, целью события может быть любой объект – включая объекты `Window` и `Document` и все объекты `Elements` элементов документа – определяющий метод с именем `addEventListener()`, с помощью которого можно регистрировать обработчики событий для этой цели. Метод `addEventListener()` принимает три аргумента. Первый – тип события, для которого регистрируется обработчик. Тип (или имя) события должен быть строкой и не должен включать префикс «on», используемый при установке свойств обработчиков событий. Вторым аргументом методу `addEventListener()` передается функция, которая должна вызываться при возникновении события указанного типа. В последнем аргументе методу `addEventListener()` передается логическое значение. Обычно в этом аргументе передается значение `false`. Если передать в нем значение `true`, функция будет зарегистрирована как *перехватывающий* обработчик и будет вызываться в другой фазе распространения события. Более подробно фаза перехвата событий будет рассматриваться в разделе 17.3.6. Спецификация со временем может измениться так, что будет допустимо опускать третий аргумент вместо того, чтобы явно передавать в нем значение `false`, но на момент написания этих строк отсутствие третьего аргумента в некоторых текущих броузерах приводила к ошибке.

Следующий фрагмент регистрирует два обработчика события «click» в элементе `<button>`. Обратите внимание на различия двух используемых приемов:

```
<button id="mybutton">Щелкни на мне</button>
<script>
var b = document.getElementById("mybutton");
b.onclick = function() { alert("Спасибо, что щелкнули на мне!"); };
b.addEventListener("click", function() { alert("Еще раз спасибо!"); }, false);
</script>
```

Вызов метода `addEventListener()` со строкой «click» в первом аргументе никак не влияет на значение свойства `onclick`. Во фрагменте, приведенном выше, щелчок на кнопке приведет к выводу двух диалогов `alert()`. Но важнее то, что метод `addEventListener()` можно вызвать несколько раз и зарегистрировать с его помощью несколько функций-обработчиков для одного и того же типа события в том же самом объекте. При появлении события в объекте будут вызваны все обработчики, зарегистрированные для этого типа события, в порядке их регистрации. Многократный вызов метода `addEventListener()` для одного и того же объекта с теми же самыми аргументами не дает никакого эффекта – функция-обработчик регистрируется только один раз и повторные вызовы не влияют на порядок вызова обработчиков.

Парным к методу `addEventListener()` является метод `removeEventListener()`, который принимает те же три аргумента, но не добавляет, а удаляет функцию-обработчик из объекта. Это часто бывает удобно, когда необходимо зарегистрировать временный обработчик события, а затем удалить его в какой-то момент. Например, при получении события «mousedown» может потребоваться зарегистрировать временный перехватывающий обработчик событий «mousemove» и «mouseup», чтобы можно было наблюдать за тем, как пользователь выполняет буксировку объектов мышью, а по событию «mouseup» эти обработчики могут удаляться. В такой ситуации реализация удаления обработчиков событий может иметь вид, как показано ниже:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

17.2.4. attachEvent()

Internet Explorer версии ниже IE9 не поддерживает методы `addEventListener()` и `removeEventListener()`. В версии IE5 и выше определены похожие методы, `attachEvent()` и `detachEvent()`.

По своему действию методы `attachEvent()` и `detachEvent()` похожи на методы `addEventListener()` и `removeEventListener()` со следующими исключениями:

- Поскольку модель событий в IE не поддерживает фазу перехвата, методы `attachEvent()` и `detachEvent()` принимают только два аргумента: тип события и функцию обработчика.
- В первом аргументе методам в IE передается имя свойства обработчика с префиксом «on», а не тип события без этого префикса. Например, методу `attachEvent()` должно передаваться имя «onclick», тогда как методу `addEventListener()` должно передаваться имя «click».
- Метод `attachEvent()` позволяет зарегистрировать одну и ту же функцию обработчика несколько раз. При возникновении события указанного типа зарегистрированная функция будет вызвана столько раз, сколько раз она была зарегистрирована.

Ниже показано, как обычно выполняется регистрация обработчика с помощью метода `addEventListener()` в броузерах, поддерживающих его, и с помощью метода `attachEvent()` в других броузерах:

```
var b = document.getElementById("mybutton");
var handler = function() { alert("Спасибо!"); };
if (b.addEventListener)
    b.addEventListener("click", handler, false);
else if (b.attachEvent)
    b.attachEvent("onclick", handler);
```

17.3. Вызов обработчиков событий

После регистрации обработчика событий веб-броузер будет вызывать его автоматически, когда в указанном объекте будет возникать событие указанного типа. В этом разделе подробно описывается порядок вызова обработчиков событий, аргументы обработчиков, контекст вызова (значение `this`), область видимости и назначение возвращаемого значения обработчика. К сожалению, некоторые из этих подробностей отличаются между IE версии 8 и ниже и другими броузерами.

Кроме того, с целью описать, как вызываются отдельные обработчики событий, в этом разделе также разъясняется, как происходит распространение событий: как единственное событие может привести к вызову нескольких обработчиков в оригинальном целевом объекте и в содержащих его элементах документа.

17.3.1. Аргумент обработчика событий

При вызове обработчика событий ему обычно (за одним исключением, о котором рассказывается ниже) передается объект события в виде единственного аргумента. Свойства объекта события содержат дополнительную информацию о событии. Свойство `type`, например, определяет тип возникшего события. В разделе 17.1 упоминалось множество других свойств объекта события для различных типов событий.

В IE версии 8 и ниже обработчикам событий, зарегистрированным установкой свойства, объект события при вызове не передается. Вместо этого объект события сохраняется в глобальной переменной `window.event`. Для переносимости обработчики событий можно оформлять, как показано ниже, чтобы они использовали переменную `window.event` при вызове без аргумента:

```
function handler(event) {
    event = event || window.event;
    // Здесь находится реализация обработчика
}
```

Объект события передается обработчикам событий, зарегистрированным с помощью метода `attachEvent()`, но они также могут использовать переменную `window.event`.

В разделе 17.2.2 говорилось, что при регистрации обработчика события посредством HTML-атрибута броузер преобразует строку с программным кодом на языке JavaScript в функцию. Броузеры, отличные от IE, создают функцию с единственным аргументом `event`. В IE создается функция, не принимающая аргументов. Если в таких функциях использовать идентификатор `event`, он будет ссылаться на `window.event`. В любом случае обработчики событий, определяемые в разметке HTML, могут ссылаться на объект события, используя идентификатор `event`.

17.3.2. Контекст обработчиков событий

Когда обработчик событий регистрируется установкой свойства, это выглядит как определение нового метода элемента документа:

```
e.onclick = function() { /* реализация обработчика */ };
```

Поэтому нет ничего удивительного, что обработчики событий вызываются (с одним исключением, касающимся IE, которое описывается ниже) как методы объектов, в которых они определены. То есть в теле обработчика событий ключевое слово `this` ссылается на цель события.

В обработчиках ключевое слово `this` ссылается на целевой объект, даже когда они были зарегистрированы с помощью метода `addEventListener()`. Однако, к сожалению, это не относится к методу `attachEvent()`: обработчики, зарегистрированные с помощью метода `attachEvent()`, вызываются как функции, и в них ключевое слово `this` ссылается на глобальный (`Window`) объект. Эту проблему можно решить следующим способом:

```
/*
 * Регистрирует указанную функцию как обработчик событий указанного типа в указанном
 * объекте. Гарантирует, что обработчик всегда будет вызываться как метод целевого объекта.
 */
function addEvent(target, type, handler) {
    if (target.addEventListener)
        target.addEventListener(type, handler, false);
    else
        target.attachEvent("on" + type,
                           function(event) {
                               // Вызвать обработчик как метод цели,
                               // и передать ему объект события
                               return handler.call(target, event);
                           });
}
```

Обратите внимание, что обработчики событий, зарегистрированные таким способом, нельзя удалить, потому что ссылка на функцию-обертку, передаваемую методу `attachEvent()`, нигде не сохраняется, чтобы ее можно было передать методу `detachEvent()`.

17.3.3. Область видимости обработчика событий

Подобно всем функциям в языке JavaScript, обработчики событий имеют лексическую область видимости. Они выполняются в той области видимости, в какой были определены, а не в той, где они были вызваны, и имеют доступ ко всем локальным переменным в этой области видимости. (Это, например, демонстрируется в функции `addEvent()`, представленной выше.)

Особый случай представляют обработчики событий, которые регистрируются посредством HTML-атрибутов. Они преобразуются в функции верхнего уровня, которые не имеют доступа ни к каким локальным переменным – только к глобальным. Но по историческим причинам они выполняются в модифицированной цепочке областей видимости. Обработчики событий, определяемые посредством HTML-атрибутов, могут использовать свойства целевого объекта, объемлющего элемента `<form>` (если таковой имеется) и объекта `Document`, как если бы они были локальными переменными. В разделе 17.2.2 было показано, как из HTML-атрибута создается функция обработчика события, программный код в которой использует цепочку областей видимости, модифицированную с помощью инструкций `with`.

HTML-атрибуты плохо подходят для включения длинных строк программного кода, и такая модифицированная цепочка областей видимости помогает сократить его. Она позволяет использовать `tagName` вместо `this.tagName`, `getElementById` вместо `document.getElementById`, а в обработчиках, привязанных к элементам документа внутри элемента `<form>`, можно ссылаться на другие элементы формы по значению атрибута `id`, используя, например, имя `zipcode` вместо `this.form.zipcode`.

С другой стороны, модифицированная цепочка областей видимости обработчика событий, определяемого с помощью HTML-атрибута, может стать источником ошибок, потому что свойства всех объектов в цепочке видимости скрывают одноименные свойства глобального объекта. Например, объект `Document` определяет (редко используемый) метод `open()`, поэтому если обработчику событий, созданному с помощью HTML-атрибута, потребуется вызвать метод `open()` объекта `Window`, он вынужден будет явно вызывать его как `window.open()`, вместо `open()`. Аналогичная (но более пагубная) проблема наблюдается при работе с формами, потому что имена и значения атрибутов `id` элементов формы определяют свойства во вмещающем элементе формы (раздел 15.9.1). То есть если, к примеру, форма содержит элемент со значением «location» атрибута `id`, все обработчики событий, созданные внутри этой формы с помощью HTML-атрибутов, должны будут использовать `window.location` вместо `location`, если им потребуется сослаться на объект `Location` окна.

17.3.4. Возвращаемые значения обработчиков

Значение, возвращаемое обработчиком события, зарегистрированным установкой свойства объекта или с помощью HTML-атрибута, следует учитывать. Обычно возвращаемое значение `false` сообщает броузеру, что он не должен выполнять действия, предусмотренные для этого события по умолчанию. Например, обработчик `onclick` кнопки отправки формы может вернуть `false`, чтобы предотвратить отправку формы броузером. (Это может пригодиться, если ввод пользователя не прошел проверку на стороне клиента.) Аналогично обработчик события `onkeypress` поля ввода может фильтровать ввод с клавиатуры, возвращая `false` при вводе недопустимых символов. (Пример 17.6 фильтрует ввод с клавиатуры именно таким способом.)

Также важно значение, возвращаемое обработчиком `onbeforeunload` объекта `Window`. Это событие генерируется, когда броузер выполняет переход на другую страницу. Если этот обработчик вернет строку, она будет выведена в модальном диалоге, предлагающем пользователю подтвердить свое желание покинуть страницу.

Важно понимать, что учитываются значения, возвращаемые обработчиками событий, только если обработчики зарегистрированы посредством установки свойств. Далее мы увидим, что обработчики, зарегистрированные с помощью `addEventListener()` или `attachEvent()` вместо этого должны вызывать метод `preventDefault()` или устанавливать свойство `returnValue` объекта события.

17.3.5. Порядок вызова

Для элемента документа или другого объекта можно зарегистрировать более одного обработчика одного и того же типа события. При возникновении этого события броузер вызовет все обработчики в порядке, определяемом следующими правилами:

- В первую очередь вызываются обработчики, зарегистрированные установкой свойства объекта или с помощью HTML-атрибута, если таковые имеются.
- Затем вызываются обработчики, зарегистрированные с помощью метода `addEventListener()`, в порядке их регистрации.[1]

---

[1] Стандарт «DOM Level 2» не определяет порядок вызова обработчиков, но все текущие броузеры вызывают обработчики в порядке их регистрации, а текущий проект стандарта «DOM Level 3» определяет именно такой порядок вызова.

- Обработчики, зарегистрированные с помощью метода `attachEvent()`, могут вызываться в произвольном порядке, поэтому ваши сценарии не должны полагаться на какой-то определенный порядок.

17.3.6. Распространение событий

Когда целью события является объект `Window` или какой-то другой самостоятельный объект (такой как `XMLHttpRequest`), броузер откликается на событие простым вызовом соответствующего обработчика в этом объекте. Однако когда целью события является объект `Document` или элемент `Element` документа, ситуация несколько осложняется.

После вызова обработчиков событий, зарегистрированных в целевом элементе, большинство событий «всплывают» вверх по дереву DOM. В результате вызываются обработчики в родителе целевого элемента. Затем вызываются обработчики, зарегистрированные в родителе родителя целевого элемента. Так продолжается, пока не будет достигнут объект `Document` и затем объект `Window`. Способность событий всплывать обеспечивает возможность реализации альтернативы множеству обработчиков, зарегистрированных в отдельных элементах документа: можно зарегистрировать единственный обработчик в общем элементе-предке и обрабатывать события в нем. Например, вместо того чтобы регистрировать обработчик события «change» в каждом элементе формы, его можно зарегистрировать в единственном элементе `<form>`.

Способностью всплывать обладает большинство событий, возникающих в элементах документа. Заметным исключением являются события «focus», «blur» и «scroll». Событие «load», возникающее в элементах, также всплывает, но оно прекращает всплывать в объекте `Document` и не достигает объекта `Window`. Событие «load» в объекте `Window` возбуждается, только когда будет загружен весь документ.

Всплытие – это третья «фаза» распространения события. Вызов обработчика события в целевом объекте – это вторая фаза. Первая фаза протекает еще до вызова обработчиков целевого объекта и называется фазой «перехвата». Напомню, что метод `addEventListener()` имеет третий аргумент, в котором принимает логическое значение. Если передать в этом аргументе значение `true`, обработчик события будет зарегистрирован как перехватывающий обработчик для вызова в первой фазе распространения события. Фаза всплытия событий реализована во всех броузерах, включая IE, и в ней участвуют все обработчики, независимо от того, как они были зарегистрированы (если только они не были зарегистрированы как перехватывающие обработчики). В фазе перехвата, напротив, участвуют только обработчики, зарегистрированные с помощью метода `addEventListener()`, когда в третьем аргументе ему было передано значение `true`. Это означает, что фаза перехвата событий недоступна в IE версии 8 и ниже, и на момент написания этих строк имела ограничения в использовании.

Фаза перехвата напоминает фазу всплытия, только событие распространяется в обратном направлении. В первую очередь вызываются перехватывающие обработчики объекта `Window`, затем вызываются перехватывающие обработчики объекта `Document`, затем обработчики объекта `body` и так далее, вниз по дереву DOM, пока не будут вызваны перехватывающие обработчики родителя целевого объекта. Перехватывающие обработчики, зарегистрированные в самом целевом объекте, не вызываются.

Наличие фазы перехвата позволяет обнаруживать события еще до того, как они достигнут своей цели. Перехватывающий обработчик может использоваться для отладки или для фильтрации событий, чтобы в комплексе с приемом отмены события, описываемом ниже, предотвратить вызов обработчиков в целевом объекте. Одной из типичных областей применения фазы перехвата является реализация буксировки элементов мышью, когда событие перемещения указателя мыши должен обрабатывать буксируемый объект, а не документ, в пределах которого осуществляется буксировка. Пример реализации приводится в примере 17.2.

17.3.7. Отмена событий

В разделе 17.3.4 говорилось, что значение, возвращаемое обработчиком события, зарегистрированным как свойство, можно использовать для отмены действий, выполняемых броузером по умолчанию в случае этого события. В броузерах, поддерживающих метод `addEventListener()`, отменить выполнение действий по умолчанию можно также вызовом метода `preventDefault()` объекта события. Однако в IE, версии 8 и ниже, тот же эффект достигается установкой свойства `returnValue` объекта события в значение `false`. В следующем фрагменте демонстрируется обработчик вымышленного события, который использует все три способа отмены события:

```
function cancelHandler(event) {
    var event = event || window.event; // Для IE

    /* Здесь выполняется обработка события */

    // Теперь отменить действие по умолчанию, связанное с событием
    if (event.preventDefault) event.preventDefault(); // Стандартный прием
    if (event.returnValue) event.returnValue = false; // IE
    return false;      // Для обработчиков, зарегистрированных как свойства
}
```

Текущий проект модуля «DOM Events» определяет в объекте `Event` свойство с именем `defaultPrevented`. Оно пока поддерживается не всеми броузерами, но суть его в том, что при обычных условиях оно имеет значение `false` и принимает значение `true` только в случае вызова метода `preventDefault()`.[1]

Отмена действий, по умолчанию связанных с событием, – это лишь одна из разновидностей отмены события. Имеется также возможность остановить распространение события. В броузерах, поддерживающих метод `addEventListener()`, объект события имеет метод `stopPropagation()`, вызов которого прерывает дальнейшее распространение события. Если в том же целевом объекте будут зарегистрированы другие обработчики этого события, то остальные обработчики все равно будут вызваны, но никакие другие обработчики событий в других объекта не будут вызваны после вызова метода `stopPropagation()`. Метод `stopPropagation()` можно вызвать в любой момент в ходе распространения события. Он действует и в фазе перехвата, и в вызове обработчика целевого объекта, и в фазе всплытия.

В IE версии 8 и ниже метод `stopPropagation()` не поддерживается. Вместо этого объект события в IE имеет свойство `cancelBubble`. Установка этого свойства в значение `true` предотвращает распространение события. (В IE версии 8 и ниже фаза перехвата не поддерживается, поэтому отменить событие можно только в фазе всплытия.)

---

[1] Объект события, определяемый в библиотеке jQuery (глава 19), вместо свойства имеет метод `defaultPrevented()`.

Текущий проект спецификации «DOM Events» определяет в объекте `Event` еще один метод – метод с именем `stopImmediatePropagation()`. Подобно методу `stopPropagation()`, он предотвращает распространение события по любым другим объектам. Но кроме того, он также предотвращает вызов любых других обработчиков событий, зарегистрированных в том же объекте. На момент написания этих строк метод `stopImmediatePropagation()` поддерживался не во всех броузерах. Некоторые библиотеки, такие как jQuery и YUI, определяют свою, переносимую, версию метода `stopImmediatePropagation()`.

17.4. События загрузки документа

Теперь, когда мы познакомились с основами обработки событий в языке JavaScript, можно приступать к изучению подробностей, касающихся отдельных категорий событий. В этом разделе мы начнем с событий, возникающих при загрузке документа.

Большинству веб-приложений совершенно необходимо, чтобы веб-броузер извещал их о моменте, когда закончится загрузка документа и он будет готов для выполнения операций над ним. Этой цели служит событие «load» в объекте `Window`, и оно уже подробно обсуждалось в главе 13, где в примере 13.5 была представлена реализация вспомогательной функции `onLoad()`. Событие «load» возбуждается только после того, как документ и все его изображения будут полностью загружены. Однако обычно сценарии можно запускать сразу после синтаксического анализа документа, до того как будут загружены изображения. Можно существенно сократить время запуска веб-приложения, если начинать выполнение сценариев по событиям, отличным от «load».

Событие «DOMContentLoaded» возбуждается, как только документ будет загружен, разобран синтаксическим анализатором, и будут выполнены все отложенные сценарии. К этому моменту изображения и сценарии с атрибутом `async` могут продолжать загружаться, но сам документ уже будет готов к выполнению операций. (Отложенные и асинхронные сценарии обсуждались в разделе 13.3.1.) Это событие впервые было введено в Firefox и впоследствии заимствовано всеми другими производителями броузеров, включая корпорацию Microsoft, которая добавила поддержку этого события в IE9. Несмотря на приставку «DOM» в имени, это событие не является частью стандарта модели событий «DOM Level 3 Events», но оно стандартизовано спецификацией HTML5.

Как описывалось в разделе 13.3.4, в ходе загрузки документа изменяется значение свойства `document.readyState`. Каждое изменение значения этого свойства в IE сопровождается событием «readystatechange» в объекте `Document`, благодаря чему в IE это событие можно использовать для определения момента появления состояния «complete». Спецификация HTML5 стандартизует событие «readystatechange», но предписывает возбуждать его непосредственно перед событием «load», поэтому не совсем понятно, в чем заключается преимущество события «readystatechange» перед «load».

В примере 17.1 определяется функция `whenReady()`, близко напоминающая функцию `onLoad()` из примера 13.5. Функция, передаваемая функции `whenReady()`, вызывается (как метод объекта `Document`) сразу, как только документ будет готов к выполнению операций. В отличие от ранее представленной функции `onLoad()`, `whenReady()` ожидает появления событий «DOMContentLoaded» и «readystatechange» и использует событие «load» только как запасной вариант, на случай если она будет задействована в старых броузерах, не поддерживающих первые два события. Функция `whenReady()` будет использоваться в некоторых сценариях, следующих далее (в этой и в других главах).

Пример 17.1. Вызов функций, когда документ будет готов к выполнению операций

```
/*
 * Передайте функции whenReady() свою функцию, и она вызовет ее (как метод
 * объекта документа), как только завершится синтаксический анализ документа
 * и он будет готов к выполнению операций.  Зарегистрированные функции
 * вызываются при первом же событии DOMContentLoaded, readystatechange или load.
 * Как только документ станет готов и будут вызваны все функции,
 * whenReady() немедленно вызовет все функции, которые были ей переданы.
 */
var whenReady = (function() { // Эта функция возвращается функцией whenReady()
    var funcs = [];           // Функции, которые должны вызываться по событию
    var ready = false;        // Получит значение true при вызове функции handler

    // Обработчик событий, который вызывается, как только документ
    // будет готов к выполнению операций
    function handler(e) {
        // Если обработчик уже вызывался, просто вернуть управление
        if (ready) return;

        // Если это событие readystatechange и состояние получило значение,
        // отличное от "complete", значит, документ пока не готов
        if (e.type==="readystatechange" && document.readyState !== "complete")
            return;

        // Вызвать все зарегистрированные функции.
        // Обратите внимание, что здесь каждый раз проверяется значение
        // свойства funcs.length, на случай если одна из вызванных функций
        // зарегистрирует дополнительные функции.
        for(var i = 0; i < funcs.length; i++)
            funcs[i].call(document);

        // Теперь можно установить флаг ready в значение true и забыть
        // о зарегистрированных функциях
        ready = true;
        funcs = null;
    }

    // Зарегистрировать обработчик handler для всех ожидаемых событий
    if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", handler, false);
        document.addEventListener("readystatechange", handler, false);
        window.addEventListener("load", handler, false);
    }
    else if (document.attachEvent) {
        document.attachEvent("onreadystatechange", handler);
        window.attachEvent("onload", handler);
    }
    // Вернуть функцию whenReady
    return function whenReady(f) {
        if (ready) f.call(document); // Вызвать функцию, если документ готов
        else funcs.push(f);          // Иначе добавить ее в очередь,
    }                                // чтобы вызвать позже.
}());
```

17.5. События мыши

С мышью связано довольно много событий. Все они перечислены в табл. 17.1. Все события мыши, кроме «mouseenter» и «mouseleave», всплывают. Для событий «click», возникающих в ссылках и кнопках отправки форм, предусматриваются действия по умолчанию, которые можно отменить. Теоретически имеется возможность отменить событие «contextmenu» и предотвратить появление контекстного меню, но некоторые броузеры имеют параметры настройки, которые делают это событие неотменяемым.

Таблица 17.1. События мыши

Тип	Описание
`click`	Высокоуровневое событие, возбуждаемое, когда пользователь нажимает и отпускает кнопку мыши или иным образом «активирует» элемент.
`contextmenu`	Отменяемое событие, возбуждаемое перед выводом контекстного меню. Текущие броузеры выводят контекстное меню по щелчку правой кнопки мыши, поэтому данное событие можно также использовать как событие «click».
`dblclick`	Возбуждается, когда пользователь выполняет двойной щелчок.
`mousedown`	Возбуждается, когда пользователь нажимает кнопку мыши.
`mouseup`	Возбуждается, когда пользователь отпускает кнопку мыши.
`mousemove`	Возбуждается, когда пользователь перемещает указатель мыши.
`mouseover`	Возбуждается, когда указатель мыши помещается над элементом. Свойство `relatedTarget` (или `fromElement` в IE) определяет элемент, с которого был перемещен указатель мыши.
`mouseout`	Возбуждается, когда указатель мыши покидает элемент. Свойство `relatedTarget` (или `toElement` в IE) определяет элемент, на который был перемещен указатель мыши.
`mouseenter`	Подобно «mouseover», но не всплывает. Впервые появилось в IE и было стандартизовано в HTML5, но пока поддерживается не всеми броузерами.
`mouseleave`	Подобно «mouseout», но не всплывает. Впервые появилось в IE и было стандартизовано в HTML5, но пока поддерживается не всеми броузерами.

Объект, передаваемый обработчикам событий от мыши, имеет свойства `clientX` и `clientY`, определяющие координаты указателя относительно окна. Чтобы преобразовать их в координаты документа, к ним необходимо добавить позиции полос прокрутки окна (как показано в примере 15.8).

Свойства `altKey`, `ctrlKey`, `metaKey` и `shiftKey` определяют состояния различных клавиш-модификаторов, которые могли удерживаться в нажатом состоянии в момент события: с их помощью можно отличать простой щелчок от щелчка с нажатой клавишей Shift, например.

Свойство `button` определяет, какая кнопка мыши удерживалась в нажатом состоянии в момент события. Однако разные броузеры записывают в это свойство разные значения, поэтому его сложно использовать переносимым способом. Подробности смотрите в справочной статье `Event` в четвертой части книги. Некоторые броузеры возбуждают событие «click» только в случае щелчка левой кнопкой. Поэтому, если потребуется обрабатывать щелчки другими кнопками, следует использовать события «mousedown» и «mouseup». Событие «contextmenu» обычно сигнализирует о том, что был выполнен щелчок правой кнопкой, но, как отмечалось выше, в обработчиках этого события не всегда бывает возможным предотвратить появление контекстного меню.

Объект события, передаваемый вместе с событием мыши, имеет еще пять характерных свойств, но они используются значительно реже, чем вышеперечисленные. Полный их перечень вы найдете в справочной статье `Event` в четвертой части книги.

В примере 17.2 демонстрируется функция `drag()`, которая при вызове из обработчика события «mousedown» позволяет пользователю буксировать мышью абсолютно позиционированные элементы документа. Функция `drag()` работает с обеими моделями событий, DOM и IE.

Функция `drag()` принимает два аргумента. Первый – буксируемый элемент. Это может быть элемент, в котором возникло событие «mousedown», и содержащий его элемент (например, можно дать пользователю возможность ухватить мышью элемент, который выглядит как заголовок окна, и буксировать содержащий его элемент, который выглядит как окно). Однако в любом случае это должен быть элемент документа, абсолютно позиционированный с помощью CSS-атрибута `position`. Второй аргумент – объект события, полученный с событием «mousedown». Ниже приводится простой пример использования функции `drag()`. В нем определяется элемент `<img>`, который пользователь может двигать мышью при нажатой клавише Shift:

```
<img src="draggable.gif"
     style="position:absolute; left:100px; top:100px;"
     onmousedown="if (event.shiftKey) drag(this, event);">
```

Функция `drag()` преобразует координаты события «mousedown» в координаты документа, чтобы определить расстояние от указателя мыши до верхнего левого угла буксируемого элемента. При этом она использует вспомогательную функцию `getScrollOffsets()` из примера 15.8. Затем функция `drag()` регистрирует обработчики событий «mousemove» и «mouseup», которые последуют за событием «mousedown». Обработчик события «mousemove» отвечает за перемещение элемента документа, а обработчик события «mouseup» – за удаление себя и обработчика события «mousemove».

Важно отметить, что обработчики событий «mousemove» и «mouseup» регистрируются как перехватывающие обработчики. Это обусловлено тем, что пользователь может перемещать мышь быстрее, чем сможет перемещаться элемент документа, в результате чего некоторые события «mousemove» могут возникать за пределами буксируемого элемента. В фазе всплытия эти события просто не будут передаваться нужным обработчикам. В отличие от стандартной модели модель событий, реализованная в IE, не поддерживает фазу перехвата, но она поддерживает специализированный метод `setCapture()`, позволяющий перехватывать собы-

тия мыши в подобных случаях. Пример наглядно демонстрирует, как действует этот метод.

Наконец, обратите внимание, что функции `moveHandler()` **и** `upHandler()` **определены внутри функции** `drag()`**. Благодаря тому, что они определены во вложенной области видимости, эти функции могут пользоваться аргументами и локальными переменными функции** `drag()`**, что существенно упрощает их реализацию.**

Пример 17.2. Буксировка элементов документа

```
/**
 * Drag.js: буксировка абсолютно позиционированных HTML-элементов.
 *
 * Этот модуль определяет единственную функцию drag(), которая должна вызываться
 * из обработчика события onmousedown. Последующие события mousemove будут вызывать
 * перемещение указанного элемента. Событие mouseup будет завершать буксировку.
 * Эта реализация действует в обеих моделях событий, стандартной и IE.
 * Использует функцию getScrollOffsets(), представленную выше в книге.
 *
 * Аргументы:
 *
 * elementToDrag: элемент, принявший событие mousedown или содержащий его элемент.
 * Этот элемент должен иметь абсолютное позиционирование. Значения его свойств style.left
 * и style.top будут изменяться при перемещении указателя мыши пользователем.
 *
 * event: объект Event, полученный обработчиком события mousedown.
 **/
function drag(elementToDrag, event) {
    // Преобразовать начальные координаты указателя мыши в координаты документа
    var scroll = getScrollOffsets(); // Вспомогательная функция, объявленная
                                     // где-то в другом месте
    var startX = event.clientX + scroll.x;
    var startY = event.clientY + scroll.y;

    // Первоначальные координаты (относительно начала документа) элемента, который
    // будет перемещаться. Так как elementToDrag имеет абсолютное позиционирование,
    // предполагается, что его свойство offsetParent ссылается на тело документа.
    var origX = elementToDrag.offsetLeft;
    var origY = elementToDrag.offsetTop;

    // Найти расстояние между точкой события mousedown и верхним левым углом элемента.
    // Это расстояние будет учитываться при перемещении указателя мыши.
    var deltaX = startX - origX;
    var deltaY = startY - origY;

    // Зарегистрировать обработчики событий mousemove и mouseup,
    // которые последуют за событием mousedown.
    if (document.addEventListener) { // Стандартная модель событий
        // Зарегистрировать перехватывающие обработчики в документе
        document.addEventListener("mousemove", moveHandler, true);
        document.addEventListener("mouseup", upHandler, true);
    }
    else if (document.attachEvent) { // Модель событий IE для IE5-8
        // В модели событий IE перехват событий осуществляется вызовом
        // метода setCapture() элемента.
        elementToDrag.setCapture();
        elementToDrag.attachEvent("onmousemove", moveHandler);
        elementToDrag.attachEvent("onmouseup", upHandler);
        // Интерпретировать потерю перехвата событий мыши как событие mouseup
        elementToDrag.attachEvent("onlosecapture", upHandler);
    }

    // Это событие обработано и не должно передаваться другим обработчикам
    if (event.stopPropagation) event.stopPropagation();  // Стандартная модель
    else event.cancelBubble = true;                      // IE

    // Предотвратить выполнение действий, предусмотренных по умолчанию.
    if (event.preventDefault) event.preventDefault();    // Стандартная модель
    else event.returnValue = false;                      // IE

    /**
     * Этот обработчик перехватывает события mousemove, возникающие
     * в процессе буксировки элемента. Он отвечает за перемещение элемента.
     **/
    function moveHandler(e) {
        if (!e) e = window.event;                        // Модель событий IE

        // Переместить элемент в позицию указателя мыши с учетом позиций
        // полос прокрутки и смещений относительно начального щелчка.
        var scroll = getScrollOffsets();
        elementToDrag.style.left = (e.clientX + scroll.x - deltaX) + "px";
        elementToDrag.style.top = (e.clientY + scroll.y - deltaY) + "px";

        // И прервать дальнейшее распространение события.
        if (e.stopPropagation) e.stopPropagation();      // Стандартная модель
        else e.cancelBubble = true;                      // IE
    }

    /**
     * Этот обработчик перехватывает заключительное событие mouseup,
     * которое завершает операцию буксировки.
     **/
    function upHandler(e) {
        if (!e) e = window.event;               // Модель событий IE

        // Удалить перехватывающие обработчики событий.
        if (document.removeEventListener) {     // Модель событий DOM
            document.removeEventListener("mouseup", upHandler, true);
            document.removeEventListener("mousemove", moveHandler, true);
        }
        else if (document.detachEvent) {        // Модель событий IE 5+
            elementToDrag.detachEvent("onlosecapture", upHandler);
            elementToDrag.detachEvent("onmouseup", upHandler);
            elementToDrag.detachEvent("onmousemove", moveHandler);
            elementToDrag.releaseCapture();
        }

        // И прервать дальнейшее распространение события.
        if (e.stopPropagation) e.stopPropagation();      // Стандартная модель
        else e.cancelBubble = true;                      // IE
    }
}
```

Следующий фрагмент демонстрирует порядок использования функции `drag()` в HTML-файле (это упрощенная версия примера 16.2 с добавленной поддержкой буксировки):

```
<script src="getScrollOffsets.js"></script> <!-- требуется функция drag()-->
<script src="Drag.js"></script>             <!-- определение drag() -->
<!-- Буксируемый элемент -->
<div style="position:absolute; left:100px; top:100px; width:250px;
            background-color: white; border: solid black;">
  <!-- "Заголовок" окна. Обратите внимание на атрибут onmousedown. -->
  <div style="background-color: gray; border-bottom: dotted black;
              padding: 3px; font-family: sans-serif; font-weight: bold;"
      onmousedown="drag(this.parentNode, event);">
    Перетащи меня <!-- Содержимое заголовка -->
  </div>
  <!-- Содержимое буксируемого элемента -->
  <p>Это тест. Проверка, проверка, проверка.</p><p>Тест</p><p>Тест</p>
</div>
```

Самым важным здесь является атрибут `onmousedown` во вложенном элементе `<div>`. Обратите внимание, что в нем используется свойство `this.parentNode`. Это говорит о том, что перемещаться будет весь контейнерный элемент.

17.6. События колесика мыши

Все современные броузеры поддерживают колесико мыши и возбуждают событие «mousewheel», когда пользователь вращает его. Броузеры часто используют колесико мыши для прокрутки документа или изменения масштаба отображения, но вы можете отменить событие «mousewheel», чтобы предотвратить выполнение этих действий по умолчанию.

Существует множество проблем совместимости, связанных с событиями «mousewheel», тем не менее вполне возможно написать программный код, действующий на всех платформах. На момент написания этих строк все броузеры, кроме Firefox, поддерживали событие с именем «mousewheel». В Firefox это событие называется «DOMMouseScroll». А в проекте спецификации «DOM Level 3 Events» предложено имя «wheel» вместо «mousewheel». Вдобавок к разным именам события различаются и имена свойств объектов, передаваемых с этими различными событиями, которые определяют величину поворота колесика. Наконец, следует отметить, что существуют также важные отличия в аппаратной реализации колесиков в разных мышах. Некоторые колесики могут вращаться только в одной плоскости, назад и вперед, а некоторые (особенно в мышах компании Apple) могут вращаться также влево и вправо (в действительности такие «колесики» являются трекболами (trackball)). Стандарт «DOM Level 3» даже включает поддержку «колесиков», которые могут вращаться в трех плоскостях – по или против часовой стрелки вдобавок к вращению вперед/назад и влево/вправо.

Объект события, передаваемый обработчику события «mousewheel», имеет свойство `wheelDelta`, определяющее величину прокрутки колесика. Один «щелчок» колесика в направлении от пользователя обычно соответствует значению 120, а один щелчок в направлении к пользователю – значению -120. В Safari и Chrome для поддержки мышей компании Apple, имеющих трекбол вместо колесика, вращающегося в одной плоскости, вдобавок к свойству wheelDelta объект события имеет свойства wheelDeltaX и wheelDeltaY, при этом значения свойств wheelDelta и wheelDeltaY всегда совпадают.

В Firefox вместо события «mousewheel» можно обрабатывать нестандартное событие «DOMMouseScroll» и использовать свойство detail объекта события вместо wheelDelta. Однако масштаб и знак изменения свойства detail отличается от wheelDelta: чтобы получить значение, эквивалентное значению свойства wheelDelta, свойство detail нужно умножить на -40.

На момент написания этих строк проект стандарта «DOM Level 3 Events» определял стандартное событие с именем «wheel» как стандартизованную версию событий «mousewheel» и «DOMMouseScroll». Согласно спецификации, объект, передаваемый обработчику события «wheel», должен иметь свойства deltaX, deltaY и deltaZ, определяющие величину прокрутки колесика в трех плоскостях. Значения этих свойств следует умножать на -120, чтобы получить значение и знак события «mousewheel».

Для всех этих типов событий объект события напоминает объекты событий мыши: он включает координаты указателя мыши и состояние клавиш-модификаторов на клавиатуре.

Пример 17.3 демонстрирует, как обрабатывать события колесика мыши и как обходить проблемы совместимости. В нем определяется функция с именем enclose(), которая обертывает в «кадр» или «видимую область» заданного размера более крупный элемент (такой как изображение) и определяет обработчик события колесика мыши, который позволяет пользователю прокручивать элемент в пределах видимой области и изменять размеры этой области. Эту функцию enclose() можно использовать, как показано ниже:

```
<script src="whenReady.js"></script>
<script src="Enclose.js"></script>
<script>
whenReady(function() {
    enclose(document.getElementById("content"),400,200,-200,-300);
});
</script>
<style>div.enclosure { border: solid black 10px; margin: 10px; }</style>
<img id="content" src="testimage.jpg"/>
```

Для обеспечения корректной работы во всех основных броузерах в примере 17.3 выполняется проверка типа броузера (раздел 13.4.5). Пример опирается на положения спецификации «DOM Level 3 Events» и включает программный код, который будет использовать событие «wheel», когда его поддержка будет реализована в броузерах.[1] Он также включает дополнительную проверку на будущее, чтобы не использовать событие «DOMMouseScroll», если Firefox начнет использовать событие «wheel» или «mousewheel». Обратите внимание, что пример 17.3 также является практическим примером управления геометрией элементов и использования приемов позиционирования средствами CSS, о которых рассказывалось в разделах 15.8 и 16.2.1.

---

[1] Довольно рискованное решение: если будущие версии стандарта не будут соответствовать проекту спецификации, имевшейся на момент написания этих строк, это может привести к нежелательным последствиям и сделать пример неработоспособным.

Пример 17.3. Обработка событий «mousewheel»

```
// Заключает элемент содержимого в фрейм, или видимую область заданной ширины
// и высоты (минимальные размеры 50 x 50). Необязательные аргументы contentX
// и contentY определяют начальное смещение содержимого относительно кадра.
// (Их значения должны быть <= 0.) Фрейму придается обработчик события mousewheel,
// который позволяет пользователю прокручивать элемент и изменять размеры фрейма.
function enclose(content, framewidth, frameheight, contentX, contentY) {
    // Эти аргументы являются не только начальными значениями: они хранят информацию
    // о текущем состоянии, изменяются и используются обработчиком события mousewheel.
    framewidth = Math.max(framewidth, 50);
    frameheight = Math.max(frameheight, 50);
    contentX = Math.min(contentX, 0) || 0;
    contentY = Math.min(contentY, 0) || 0;

    // Создать фрейм и определить для него стили и имя класса CSS
    var frame = document.createElement("div");
    frame.className = "enclosure"; // Благодаря этому можно определять стили
                                   // в таблицах стилей
    frame.style.width = framewidth + "px";      // Установить размеры фрейма.
    frame.style.height = frameheight + "px";
    frame.style.overflow = "hidden";            // Без полос прокрутки
    frame.style.boxSizing = "border-box";       // модель border-box упрощает
    frame.style.webkitBoxSizing = "border-box"; // вычисление новых размеров
    frame.style.MozBoxSizing = "border-box";    // фрейма.

    // Добавить фрейм в документ и поместить в него элемент elt.
    content.parentNode.insertBefore(frame, content);
    frame.appendChild(content);

    // Координаты элемента относительно фрейма
    content.style.position = "relative";
    content.style.left = contentX + "px";
    content.style.top = contentY + "px";

    // Необходимо решить некоторые проблемы совместимости броузеров
    var isMacWebkit = (navigator.userAgent.indexOf("Macintosh") !== -1 &&
                       navigator.userAgent.indexOf("WebKit") !== -1);
    var isFirefox = (navigator.userAgent.indexOf("Gecko") !== -1);

    // Зарегистрировать обработчики событий mousewheel.
    frame.onwheel = wheelHandler;       // Для будущих броузеров
    frame.onmousewheel = wheelHandler; // Для большинства текущих броузеров
    if (isFirefox)                      // Только Firefox
        frame.addEventListener("DOMMouseScroll", wheelHandler, false);

    function wheelHandler(event) {
        var e = event || window.event; // Объект события, стандартный или IE

        // Получить величину прокрутки из объекта события, проверив свойства объекта
        // события wheel, mousewheel (в обоих, 2-мерном и 1-мерном вариантах)
        // и DOMMouseScroll в Firefox.Масштабировать значение так, чтобы один «щелчок»
        // колесика соответствовал 30 пикселам. Если какой-либо из броузеров в будущем
        // будет возбуждать оба типа событий, "wheel" и "mousewheel", одно и то же
        // событие будет обрабатываться дважды. Но будем надеяться, что отмена
        // события wheel будет предотвращать возбуждение события mousewheel.
        var deltaX = e.deltaX*-30 ||      // событие wheel
```

```
                 e.wheelDeltaX/4 ||    // mousewheel
                 0;                    // свойство не определено
    var deltaY = e.deltaY*-30 ||       // событие wheel
                 e.wheelDeltaY/4 ||    // событие mousewheel в Webkit
(e.wheelDeltaY===undefined &&          // Если нет 2-мерного свойства,
                 e.wheelDelta/4) ||    // использовать 1-мерное свойство
                    e.detail*-10 ||    // событие DOMMouseScroll в Firefox
                              0;       // свойство не определено

    // Большинство броузеров генерируют одно событие со значением 120
    // на один щелчок колесика. Однако похоже, что мыши компании Apple
    // более чувствительные, и значения свойств delta часто оказываются
    // в несколько раз больше 120, по крайней мере, для Apple Mouse.
    // Использовать прием проверки типа броузера, чтобы решить эту проблему.
    if (isMacWebkit) {
        deltaX /= 30;
        deltaY /= 30;
    }

    // Если мы когда-нибудь будем получать событие mousewheel или wheel в (будущих
    // версиях) Firefox, можно отказаться от обработки события DOMMouseScroll.
    if (isFirefox && e.type !== "DOMMouseScroll")
        frame.removeEventListener("DOMMouseScroll", wheelHandler, false);

    // Получить текущие размеры элемента содержимого
    var contentbox = content.getBoundingClientRect();
    var contentwidth = contentbox.right - contentbox.left;
    var contentheight = contentbox.bottom - contentbox.top;

    // Если удерживается нажатой клавиша Alt, изменить размеры фрейма
    if (e.altKey) {
        if (deltaX) {
            framewidth -= deltaX; // Новая ширина фрейма, но не больше,
            framewidth = Math.min(framwidth, contentwidth); // чем ширина
            framewidth = Math.max(framewidth,50);          // содержимого
            frame.style.width = framewidth + "px";         // и не меньше 50
        }
        if (deltaY) {
            frameheight -= deltaY; // То же для высоты фрейма
            frameheight = Math.min(frameheight, contentheight);
            frameheight = Math.max(frameheight-deltaY, 50);
            frame.style.height = frameheight + "px";
        }
    }
    else { // Клавиша Alt не нажата, прокрутить содержимое в фрейме
        if (deltaX) {
            // Прокручивать не больше, чем
            var minoffset = Math.min(framewidth-contentwidth, 0);
            // Сумма deltaX и contentX, но не меньше, чем minoffset
            contentX = Math.max(contentX + deltaX, minoffset);
            contentX = Math.min(contentX, 0);     // или больше 0
            content.style.left = contentX + "px"; // Новое смещение
        }
        if (deltaY) {
            var minoffset = Math.min(frameheight - contentheight, 0);
            // Сумма deltaY и contentY, но не меньше, чем minoffset
```

```
                contentY = Math.max(contentY + deltaY, minoffset);
                contentY = Math.min(contentY, 0);     // или больше 0
                content.style.top = contentY + "px"; // Новое смещение.
            }
        }

        // Не позволять всплывать этому событию. Предотвратить выполнение действий
        // по умолчанию. Это позволит предотвратить прокрутку содержимого документа
        // в окне броузера. Будем надеяться, что вызов preventDefault() для события wheel
        // также предотвратит возбуждение дублирующего события mousewheel.
        if (e.preventDefault) e.preventDefault();
        if (e.stopPropagation) e.stopPropagation();
        e.cancelBubble = true; // модель событий IE
        e.returnValue = false; // модель событий IE
        return false;
    }
}
```

17.7. События механизма буксировки (drag-and-drop)

В примере 17.2 было показано, как реализовать операцию буксировки элементов мышью. Она позволяет перетаскивать и оставлять элементы в пределах веб-страницы, но истинная буксировка – это нечто иное. Буксировка (drag-and-drop, или DnD) – это интерфейс взаимодействия с пользователем, позволяющий перемещать данные между «источником» и «приемником», которые могут находиться в одном или в разных приложениях. Буксировка (DnD) – это сложный механизм организации взаимодействий между человеком и машиной, и прикладные интерфейсы, реализующие поддержку буксировки, всегда отличались высокой сложностью:

- **Они должны взаимодействовать с операционной системой, чтобы обеспечить возможность взаимодействий между различными приложениями.**
- **Они должны поддерживать такие операции передачи данных, как «перемещение», «копирование» и «создание ссылки», позволять источникам и приемникам ограничивать множество допустимых операций, а также давать пользователям возможность выбирать (обычно с помощью клавиш-модификаторов) операцию из разрешенного набора.**
- **Они должны предоставлять источнику способ определять ярлык или изображение, которое будет отображаться в процессе буксировки.**
- **Они должны предоставлять механизм событий для посылки извещений источнику и приемнику в процессе буксировки.**

Корпорация Microsoft реализовала прикладной интерфейс механизма буксировки в ранних версиях IE. Он был не очень хорошо продуман и плохо документирован, тем не менее другие производители броузеров попытались скопировать его, а спецификация HTML5 стандартизовала некоторый API, напоминающий прикладной интерфейс в IE, и добавила новые особенности, делающие этот API более простым в использовании. На момент написания этих строк данный новый, более простой в использовании API буксировки еще не был реализован, поэтому

в этом разделе будет рассматриваться прикладной интерфейс в IE, как взятый за основу стандартом HTML5.

Прикладной интерфейс механизма буксировки в IE довольно сложен в использовании, а различия между реализациями в текущих броузерах не позволяют использовать наиболее сложные части API переносимым способом. Тем не менее он дает возможность веб-приложениям участвовать в операциях буксировки подобно обычным приложениям. Броузеры всегда позволяли выполнять простейшие операции буксировки. Если выделить текст в веб-броузере, его легко можно отбуксировать в текстовый процессор. А если выделить URL-адрес в текстовом процессоре, его можно отбуксировать в броузер, чтобы открыть страницу с этим адресом. В этом разделе будет показано, как создавать собственные источники, которые позволят перемещать данные, не являющиеся текстом, и собственные приемники, откликающиеся на попытки оставить в них данные некоторым способом, помимо простого их отображения.

Механизм буксировки всегда опирался на события, поэтому в JavaScript API реализовано два множества событий: события из первого множества возбуждаются в источнике данных, а из второго – в приемнике. Все обработчики событий буксировки получают объект события, подобный объекту события мыши, с дополнительным свойством `dataTransfer`. Это свойство ссылается на объект `DataTransfer`, определяющий методы и свойства прикладного интерфейса механизма буксировки.

События, возбуждаемые в источнике, относительно просты, поэтому начнем с них. Источником для механизма буксировки является любой элемент документа, имеющий HTML-атрибут `draggable`. Когда пользователь начинает перемещать указатель мыши с нажатой кнопкой над элементом-источником, броузер не выделяет содержимое элемента, а возбуждает в нем событие «dragstart». Обработчик этого события должен вызвать метод `dataTransfer.setData()`, чтобы определить данные (и тип этих данных), доступные в источнике. (Когда будет реализован новый HTML5 API, вместо этого метода нужно будет вызывать метод `dataTransfer.items.add()`.) Обработчику также может потребоваться установить свойство `dataTransfer.effectAllowed`, чтобы определить тип операции – «перемещение», «копирование» или «создание ссылки» – поддерживаемой источником, и, возможно, необходимо будет вызвать метод `dataTransfer.setDragImage()` или `dataTransfer.addElement()` (в броузерах, поддерживающих эти методы), чтобы определить изображение или элемент документа, который будет использоваться для визуального представления перемещаемых данных.

В процессе буксировки броузер возбуждает в источнике данных события «drag». Обработчик этого события можно использовать для изменения перемещаемого ярлыка или перемещаемых данных, но в общем случае нет никакой необходимости регистрировать обработчики событий «drag».

Когда выполняется сброс, возбуждается событие «dragend». Если источник поддерживает операцию «перемещения», он должен проверить свойство `dataTransfer.dropEffect`, чтобы убедиться, действительно ли была выполнена операция перемещения. Если это так, данные, перемещенные в другое место, следует удалить из источника.

Событие «dragstart» является единственным, обработку которого необходимо реализовать в простейшем источнике данных. Реализация такого источника представлена в примере 17.4. Он отображает текущее время в формате «hh:mm» в элементе `<span>` и обновляет время раз в минуту. Если бы это было все, что реализует пример, пользователь мог бы просто выделить отображаемый текст и отбуксировать его. Но этот пример превращает часы в источник данных для механизма буксировки, устанавливая свойство `draggable` элемента часов в значение `true` и определяя функцию-обработчик `ondragstart`. Обработчик вызывает метод `dataTransfer.setData()`, чтобы определить перемещаемые данные – строку с полной информацией о текущем времени (включая дату, секунды и информацию о часовом поясе). Он также вызывает `dataTransfer.setDragIcon()`, чтобы определить изображение (ярлык с изображением часов), которое будет перемещаться в процессе буксировки.

Пример 17.4. Источник данных для механизма буксировки

```
<script src="whenReady.js"></script>
<script>
whenReady(function() {
    var clock = document.getElementById("clock"); // Элемент часов
    var icon = new Image();                        // Буксируемое изображение
    icon.src = "clock-icon.png";                   // URL-адрес изображения

    // Отображает время раз в минуту
    function displayTime() {
        var now = new Date();                      // Получить текущее время
        var hrs = now.getHours(), mins = now.getMinutes();
        if (mins < 10) mins = "0" + mins;
        clock.innerHTML = hrs + ":" + mins;        // Отобразить текущее время
        setTimeout(displayTime, 60000);            // Запустить через 1 минуту
    }
    displayTime();

    // Сделать часы доступными для буксировки.
    // То же самое можно сделать с помощью HTML-атрибута:
    // <span draggable="true">...
    clock.draggable = true;

    // Обработчики событий
    clock.ondragstart = function(event) {
        var event = event || window.event; // Для совместимости с IE

        // Свойство dataTransfer является ключом к drag-and-drop API
        var dt = event.dataTransfer;

        // Сообщить броузеру, какие данные будут буксироваться.
        // Если конструктор Date() вызывается как функция, он возвращает
        // строку с полной информацией о текущем времени
        dt.setData("Text", Date() + "\n");

        // Определить ярлык, который будет служить визуальным представлением перемещаемой
        // строки, в броузерах, поддерживающих эту возможность. Без этого для визуального
        // представления броузер мог бы использовать изображение текста в часах.
        if (dt.setDragImage) dt.setDragImage(icon, 0, 0);
    };
});
</script>
<style>
```

```
#clock { /* Придать часам привлекательный внешний вид */
    font: bold 24pt sans; background: #ddf; padding: 10px;
    border: solid black 2px; border-radius: 10px;
}
</style>
<h1>Drag timestamps from the clock</h1>
<span id="clock"></span> <!-- Здесь отображается время -->
<textarea cols=60 rows=20></textarea> <!-- Сюда можно сбросить строку -->
```

Приемники буксируемых данных сложнее в реализации, чем источники. Приемником может быть любой элемент документа: чтобы создать приемник, не требуется устанавливать HTML-атрибуты, как при создании источников, – достаточно просто определить соответствующие обработчики событий. (Однако с реализацией нового прикладного интерфейса буксировки, определяемого стандартом HTML5, вместо некоторых обработчиков событий, описываемых ниже, в элементе-приемнике необходимо будет установить атрибут `dropzone`.) В приемнике возбуждается четыре события. Когда буксируемый объект оказывается над элементом документа, броузер возбуждает в этом элементе событие «dragenter». Определить, содержит ли буксируемый объект данные в формате, понятном приемнику, можно с помощью свойства `dataTransfer.types`. (При этом может также потребоваться проверить свойство `dataTransfer.effectAllowed`, чтобы убедиться, что источник и приемник поддерживают выполняемую операцию: перемещение, копирование или создание ссылки.) В случае успешного прохождения этих проверок элемент-приемник должен дать знать пользователю и броузеру, что он может принять буксируемый объект. Обратную связь с пользователем можно реализовать в виде изменения цвета рамки или фона. Самое интересное: приемник сообщает броузеру, что он готов принять буксируемый объект, отменяя это событие.

Если элемент не отменит событие «dragenter», переданное ему броузером, броузер не будет считать этот элемент приемником для данной операции буксировки и не будет передавать ему дополнительные события. Но если приемник отменит событие «dragenter», броузер будет посылать ему события «dragover», пока пользователь будет буксировать объект над приемником. Интересно (снова) отметить, что приемник должен обрабатывать и отменять все эти события, чтобы сообщить, что он по-прежнему готов принять буксируемый объект. Если приемнику потребуется сообщить, что он поддерживает только операцию перемещение, копирования или создания ссылки, он должен устанавливать свойство `dataTransfer.dropEffect` в обработчике события «dragover».

Если пользователь переместит буксируемый объект за границы приемника, который сообщал о своей готовности принять объект отменой событий, в приемнике будет возбуждено событие «dragleave». Обработчик этого события должен восстановить прежний цвет рамки или фона элемента или отменить любые другие визуальные изменения, выполненные в ответ на событие «dragenter». К сожалению, оба события, «dragenter» и «dragleave», всплывают; и если приемник имеет вложенные элементы, будет сложно отличить, говорит ли событие «dragleave» о том, что указатель мыши с буксируемым объектом вышел за границы элемента-приемника или что он вышел за границы вложенного элемента внутри приемника.

Наконец, если пользователь сбросит буксируемый объект над приемником, в приемнике будет возбуждено событие «drop». Обработчик этого события должен получить перемещаемые данные с помощью `dataTransfer.getData()` и выполнить над ними соответствующие операции. Если пользователь сбросит над приемником один или более файлов, свойство `dataTransfer.files` будет содержать объект, подобный массиву, с объектами `File`. (Работа с этим свойством демонстрируется в примере 18.11.) После реализации нового HTML5 API обработчики события «drop» должны будут выполнить обход элементов массива `dataTransfer.items[]`, чтобы обработать данные, которые могут быть или не быть файлами.

Пример 17.5 демонстрирует, как превратить в приемники элементы `<ul>` и как превратить вложенные в них элементы `<li>` в источники. Этот пример является демонстрацией концепции ненавязчивого JavaScript. Он отыскивает элементы списка `<ul>`, атрибут `class` которых включает класс «dnd», и регистрирует в них обработчики событий буксировки. Обработчики событий превращают список в приемник: любой текст, сброшенный над таким списком, будет преобразован в новый элемент списка и добавлен в конец. Обработчики событий также обслуживают ситуации перемещения элементов внутри списка и делают текст каждого элемента списка доступным для буксировки. Обработчики событий источников поддерживают операции копирования и перемещения и удаляют элементы списка, которые были сброшены в ходе выполнения операции перемещения. (Обратите внимание, что не все броузеры обеспечивают переносимую поддержку операции перемещения.)

Пример 17.5. Список как приемник и источник

```
/*
 * Прикладной программный интерфейс механизма буксировки весьма сложен, его реализации
 * в разных броузерах не являются полностью совместимыми. В своей основе этот пример
 * реализован правильно, но все броузеры немного отличаются друг от друга,
 * и каждый из них имеет свои уникальные особенности. В данном примере не делается
 * попыток реализовать обходные решения, характерные для отдельных броузеров.
 */
whenReady(function() { // Вызовет эту функцию, когда документ будет загружен

    // Отыскать все элементы <ul class='dnd'> и вызвать функцию dnd() для них
    var lists = document.getElementsByTagName("ul");
    var regexp = /\bdnd\b/;
    for(var i = 0; i < lists.length; i++)
        if (regexp.test(lists[i].className)) dnd(lists[i]);

    // Добавляет обработчики событий буксировки в элемент списка
    function dnd(list) {
        var original_class = list.className; // Сохранить начальный CSS-class
        var entered = 0;                     // Вход и выход за границы

    // Этот обработчик вызывается, когда буксируемый объект оказывается над списком.
    // Он проверяет, содержит ли буксируемый объект данные в поддерживаемом формате,
    // и, если это так, отменяет событие, чтобы сообщить, что список готов
    // принять объект. В этом случае он также подсвечивает элемент-приемник,
    // чтобы показать пользователю, что готов к приему данных.
    list.ondragenter = function(e) {
        e = e || window.event;        // Объект события, стандартный или IE
        var from = e.relatedTarget;

        // События dragenter и dragleave всплывают, из-за чего сложнее определить,
        // когда следует подсвечивать элемент, а когда снимать подсветку в случаях,
        // подобных этому, где элемент <ui> содержит дочерние элементы <li>.
    // В броузерах, поддерживающих свойство relatedTarget, эту проблему можно решить.
    // В других броузерах приходится считать пары событий входа/выхода.

    // Если указатель мыши оказался над списком, переместившись из-за его пределов,
    // или он оказался над списком впервые, необходимо выполнить некоторые операции
    entered++;
    if ((from && !ischild(from, list)) || entered == 1) {
        // Вся информация о буксируемом объекте находится в объекте dataTransfer
        var dt = e.dataTransfer;

        // Объект dt.types содержит список типов, или форматов, в которых доступны
        // буксируемые данные. Спецификация HTML5 требует, чтобы свойство types имело
        // метод contains(). В некоторых броузерах это свойство является массивом
        // с методом indexOf. В IE версии 8 и ниже оно просто отсутствует.
        var types = dt.types; // В каких форматах доступны данные

        // Если информация о типах отсутствует или данные доступны в простом
        // текстовом формате, подсветить список, чтобы показать пользователю, что он
        // готов принять данные, и вернуть false, чтобы известить о том же и броузер.
        if (!types ||                                           // IE
             (types.contains && types.contains("text/plain")) || //HTML5
             (types.indexOf && types.indexOf("text/plain")!=-1)) //Webkit
        {
            list.className = original_class + " droppable";
            return false;
        }
        // Если тип данных не поддерживается, мы не сможем принять их
        return;     // без отмены
    }
    return false;  // Если это не первое вхождение, мы по-прежнему готовы
};

// Этот обработчик вызывается в ходе буксировки объекта над списком.
// Этот обработчик должен быть определен, и он должен возвращать false,
// иначе сброс объектов будет невозможен.
list.ondragover = function(e) { return false; };

// Этот обработчик вызывается, когда буксируемый объект выходит за границы списка
// или за границы одного из его дочерних элементов. Если объект действительно
// покидает границы списка (а не границы одного из его элементов),
// то нужно снять подсветку списка.
list.ondragleave = function(e) {
    e = e || window.event;
    var to = e.relatedTarget;

    // Если буксируемый объект покидает границы списка или если количество выходов
    // за границы совпадает с количеством входов, следует снять подсветку списка
    entered--;
    if ((to && !ischild(to,list)) || entered <= 0) {
        list.className = original_class;
        entered = 0;
    }
    return false;
};

// Этот обработчик вызывается, когда происходит сброс объекта.
// Он извлекает сброшенный текст и превращает его в новый элемент <li>
list.ondrop = function(e) {
    e = e || window.event;          // Получить объект события

    // Получить сброшенные данные в текстовом формате.
    // "Text" - это псевдоним для "text/plain".
    // IE не поддерживает "text/plain", поэтому здесь используется "Text".
    var dt = e.dataTransfer;        // объект dataTransfer
    var text = dt.getData("Text"); // Получить данные в текстовом формате.

    // Если был получен некоторый текст, превратить его в новый элемент
    // списка и добавить в конец.
    if (text) {
        var item = document.createElement("li"); // Создать новый <li>
        item.draggable = true;                   // Сделать буксируемым
        item.appendChild(document.createTextNode(text)); // Добавить текст
        list.appendChild(item);                  // Добавить в список

        // Восстановить первоначальный стиль списка и сбросить счетчик entered
        list.className = original_class;
        entered = 0;

        return false;
    }
};

// Сделать все элементы списка буксируемыми
var items = list.getElementsByTagName("li");
for(var i = 0; i < items.length; i++)
    items[i].draggable = true;

// И зарегистрировать обработчики для поддержки буксировки элементов списка.
// Обратите внимание, что мы поместили эти обработчики в список и ожидаем,
// что события будут всплывать вверх от элементов списка.

// Этот обработчик вызывается, когда буксировка начинается внутри списка.
list.ondragstart = function(e) {
    var e = e || window.event;
    var target = e.target || e.srcElement;
    // Если всплыло событие от элемента, отличного от <li>, игнорировать его
    if (target.tagName !== "LI") return false;
    // Получить важный объект dataTransfer
    var dt = e.dataTransfer;
    // Сохранить данные и указать информацию об их формате
    dt.setData("Text", target.innerText || target.textContent);
    // Сообщить, что поддерживаются операции копирования и перемещения
    dt.effectAllowed = "copyMove";
};

// Этот обработчик вызывается после успешного сброса
list.ondragend = function(e) {
    e = e || window.event;
    var target = e.target || e.srcElement;

    // Если выполнялась операция перемещения, удалить элемент списка.
    // В IE8 это свойство будет иметь значение "none", если явно
    // не установить его в значение "move" в обработчике ondrop выше.
```

```
            // Но принудительная установка в значение "move" для IE будет
            // препятствовать другим броузерам дать пользователю возможность
            // выбирать между операцииям перемещения и копирования.
            if (e.dataTransfer.dropEffect === "move")
                target.parentNode.removeChild(target);
        }

        // Вспомогательная функция, используемая в обработчиках ondragenter и ondragleave.
        // Возвращает true, если элемент a является дочерним по отношению к элементу b.
        function ischild(a,b) {
            for(; a; a = a.parentNode) if (a === b) return true;
                return false;
            }
        }
    });
```

17.8. События ввода текста

Броузеры поддерживают три старых события ввода с клавиатуры. События «keydown» и «keyup» являются низкоуровневыми событиями и рассматриваются в следующем разделе. Однако событие «keypress» является высокоуровневым, сообщающим, что был введен печатаемый символ. Проект спецификации «DOM Level 3 Events» определяет более обобщенное событие «textinput», генерируемое в ответ на ввод текста, независимо от того, каким способом он был введен (например, с клавиатуры, копированием данных из буфера обмена или с помощью операции буксировки, методом ввода текста на азиатских языках или с помощью системы распознавания речи или распознавания рукописного текста). На момент написания этих строк событие «textinput» еще не поддерживалось, но броузеры на основе ядра Webkit поддерживали очень похожее событие «textInput» (с заглавной буквой «I»).

С предлагаемым событием «textinput» и реализованным в настоящее время событием «textInput» передается простой объект события, имеющий свойство `data`, хранящее введенный текст. (Другое предлагаемое спецификацией свойство, `inputMethod`, должно определять источник ввода, но оно пока не реализовано.) В случае ввода с клавиатуры свойство `data` обычно будет содержать единственный символ, но при вводе из других источников в нем может содержаться множество символов.

Объект события, передаваемый с событием «keypress», имеет более сложную организацию. Событие «keypress» представляет ввод единственного символа. Этот символ содержится в объекте события в виде числового значения кодового пункта Юникода и, чтобы преобразовать его в строку, необходимо использовать метод `String.fromCharCode()`. В большинстве броузеров кодовый пункт введенного символа сохраняется в свойстве `keyCode` объекта события. Однако по историческим причинам в Firefox вместо него используется свойство `charCode`. В большинстве броузеров событие «keypress» возбуждается только при вводе печатаемого символа. Однако в Firefox событие «keypress» возбуждается также для непечатаемых символов. Чтобы отличить эти два случая (и проигнорировать непечатаемые символы), можно проверить, существует ли свойство `charCode` объекта события и содержит ли оно значение 0.

События «textinput», «textInput» и «keypress» можно отменить, чтобы предотвратить ввод символа. То есть эти события можно использовать для фильтрации ввода. Например, можно предотвратить ввод алфавитных символов в поле, предназначенное для ввода числовых данных. В примере 17.6 демонстрируется модуль на языке JavaScript, реализующий такого рода фильтрацию. Он отыскивает элементы `<input type=text>` с дополнительным (нестандартным) атрибутом `data-allowed-chars`. Регистрирует обработчики событий «textinput», «textInput» и «keypress» для всех найденных элементов и ограничивает возможность ввода символами, перечисленными в атрибуте `data-allowed-chars`. Первый комментарий в начале примера 17.6 включает образец разметки HTML, использующей этот модуль.

Пример 17.6. Фильтрация ввода пользователя

```
/**
 * InputFilter.js: фильтрация ввода для элементов <input>
 *
 * Этот модуль отыскивает все элементы <input type="text"> в документе, имеющие
 * атрибут "data-allowed-chars". Регистрирует обработчики событий keypress, textInput
 * и textinput для этих элементов, чтобы ограничить набор допустимых для ввода символов,
 * чтобы разрешить вводить только символы, указанные в атрибуте. Если элемент <input>
 * также имеет атрибут "data-messageid", значение этого атрибута интерпретируется как id
 * другого элемента документа. Если пользователь вводит недопустимый символ, элемент
 * с указанным id делается видимым. Если пользователь вводит допустимый символ, элемент
 * с сообщением скрывается. Данный элемент с сообщением предназначается для вывода
 * пояснений, почему ввод пользователя был отвергнут. Его оформление необходимо
 * реализовать с помощью CSS так, чтобы изначально он был невидим.
 *
 * Ниже приводится образец разметки HTML, использующей этот модуль.
 * Zipcode: <input id="zip" type="text"
 *                 data-allowed-chars="0123456789" data-messageid="zipwarn">
 * <span id="zipwarn" style="color:red;visibility:hidden">Только цифры</span>
 *
 * Этот модуль полностью реализован в ненавязчивом стиле: он не определяет
 * никаких переменных в глобальном пространстве имен.
 */
whenReady(function () { // Вызовет эту функцию, когда документ будет загружен
    // Отыскать все элементы <input>
    var inputelts = document.getElementsByTagName("input");
    // Обойти их в цикле
    for(var i = 0 ; i < inputelts.length; i++) {
        var elt = inputelts[i];
        // Пропустить элементы, не являющиеся текстовыми полями ввода
        // и не имеющие атрибута data-allowed-chars.
        if (elt.type != "text" || !elt.getAttribute("data-allowed-chars"))
            continue;

        // Зарегистрировать наш обработчик события в этом элементе input
        // keypress - старое событие и реализовано во всех броузерах.
        // textInput (смешанный регистр символов) поддерживается в Safari
        // и Chrome с 2010 года.
        // textinput (все символы строчные) - версия проекта
        // стандарта "DOM Level 3 Events".
        if (elt.addEventListener) {
            elt.addEventListener("keypress", filter, false);
            elt.addEventListener("textInput", filter, false);
            elt.addEventListener("textinput", filter, false);
        }
        // textinput не поддерживается версиями IE, в которых не реализован
        // метод addEventListener()
        else {
            elt.attachEvent("onkeypress", filter);
        }
    }

    // Обработчик событий keypress и textInput, фильтрующий ввод пользователя
    function filter(event) {
        // Получить объект события и целевой элемент target
        var e = event || window.event;          // Модель стандартная или IE
        var target = e.target || e.srcElement;  // Модель стандартная или IE
        var text = null;                        // Введенный текст

        // Получить введенный символ или текст
        if (e.type === "textinput" || e.type === "textInput") text = e.data;
        else {                                  // Это было событие keypress
            // Введенный печатаемый символ в Firefox сохраняется в свойстве charCode
            var code = e.charCode || e.keyCode;

            // Если была нажата какая-либо функциональная клавиша, не фильтровать ее
            if (code < 32 ||           // Управляющий символ ASCII
                e.charCode == 0 ||     // Функциональная клавиша (в Firefox)
                e.ctrlKey || e.altKey) // Удерживаемая клавиша-модификатор
                return;                // Не фильтровать это событие

            // Преобразовать код символа в строку
            var text = String.fromCharCode(code);
        }

        // Отыскать необходимую нам информацию в этом элементе input
        var allowed = target.getAttribute("data-allowed-chars"); // Допустимые символы
        var messageid = target.getAttribute("data-messageid");   // Сообщение id
        if (messageid) // Если указано значение id, получить элемент
            var messageElement = document.getElementById(messageid);

        // Обойти в цикле символы во введенном тексте
        for(var i = 0; i < text.length; i++) {
            var c = text.charAt(i);
            if (allowed.indexOf(c) == -1) { // Недопустимый символ?
                // Отобразить элемент с сообщением, если указан
                if (messageElement) messageElement.style.visibility="visible";

                // Отменить действия по умолчанию, чтобы предотвратить вставку текста
                if (e.preventDefault) e.preventDefault();
                if (e.returnValue) e.returnValue = false;
                return false;
            }
        }

        // Если все символы оказались допустимыми, скрыть элемент
        // с сообщением, если он был указан.
        if (messageElement) messageElement.style.visibility = "hidden";
    }
});
```

События «keypress» и «textinput» генерируются непосредственно перед фактической вставкой нового текста в элемент документа, обладающий фокусом ввода, благодаря чему обработчики этих событий могут предотвратить вставку текста, отменив событие. Броузеры также реализуют событие «input», которое возбуждается после вставки текста в элемент. Это событие нельзя отменить и соответствующий ему объект события не содержит информации о вставленном тексте – оно просто извещает о том, что текстовое содержимое элемента изменилось. Если, к примеру, потребуется обеспечить ввод только символов в верхнем регистре, можно определить обработчик события «input», как показано ниже:

```
SURNAME: <input type="text" oninput="this.value = this.value.toUpperCase();">
```

Событие «input» стандартизовано в спецификации HTML5 и поддерживается всеми современными броузерами, кроме IE. Похожего эффекта в IE можно добиться, обнаруживая изменение значения свойства `value` **текстового элемента ввода с помощью нестандартного события «propertychange». В примере 17.7 демонстрируется, как можно реализовать преобразование всех вводимых символов в верхний регистр переносимым образом.**

Пример 17.7. Использование события «propertychange» для определения факта ввода текста

```
function forceToUpperCase(element) {
    if (typeof element === "string") element=document.getElementById(element);
    element.oninput = upcase;
    element.onpropertychange = upcaseOnPropertyChange;

    // Простой случай: обработчик события input
    function upcase(event) { this.value = this.value.toUpperCase(); }
    // Сложный случай: обработчик события propertychange
    function upcaseOnPropertyChange(event) {
        var e = event || window.event;
        // Если значение свойства value изменилось
        if (e.propertyName === "value") {
            // Удалить обработчик onpropertychange, чтобы избежать рекурсии
            this.onpropertychange = null;
            // Преобразовать все символы в верхний регистр
            this.value = this.value.toUpperCase();
            // И восстановить обработчик события propertychange
            this.onpropertychange = upcaseOnPropertyChange;
        }
    }
}
```

17.9. События клавиатуры

События «keydown» и «keyup» возбуждаются, когда пользователь нажимает или отпускает клавишу на клавиатуре. Они генерируются для клавиш-модификаторов, функциональных клавиш и алфавитно-цифровых клавиш. Если пользователь удерживает клавишу нажатой настолько долго, что включается режим автоповтора, будет сгенерировано множество событий «keydown», прежде чем появится событие «keyup».

Объект события, соответствующий этим событиям, имеет свойство `keyCode` с числовым значением, которое определяет нажатую клавишу. Для клавиш, генерирующих печатаемые символы, в общем случае свойство `keyCode` содержит кодовый пункт Юникода, соответствующий основному символу, изображенному на клавише. Клавиши с буквами всегда генерируют значения `keyCode`, соответствующие символам в верхнем регистре, независимо от состояния клавиши Shift, поскольку именно такие символы изображены на клавишах. Аналогично цифровые клавиши всегда генерируют значения `keyCode`, соответствующие цифровым символам, изображенным на клавишах, даже если при этом вы удерживали нажатой клавишу Shift, чтобы ввести знак препинания. Для клавиш, не соответствующих печатаемым символам, свойство `keyCode` будет иметь некоторое другое значение. Эти значения свойства `keyCode` никогда не были стандартизованы. Однако в разных броузерах они отличаются не настолько сильно, чтобы нельзя было обеспечить переносимость. Это демонстрирует пример 17.8, включающий реализацию отображения значений `keyCode` в имена функциональных клавиш.

Подобно объектам событий мыши, объекты событий клавиатуры имеют свойства `altKey`, `ctrlKey`, `metaKey` и `shiftKey`, которые получают значение `true`, если в момент возникновения события удерживалась нажатой соответствующая клавиша-модификатор.

События «keydown» и «keyup», а также свойство `keyCode` используются уже более десяти лет, но они так и не были стандартизованы. Проект стандарта «DOM Level 3 Events» стандартизует типы «keydown» и «keyup» событий, но не стандартизует свойство `keyCode`. Вместо этого он определяет новое свойство `key`, которое должно содержать название клавиши в виде строки. Если клавиша соответствует печатаемому символу, свойство `key` должно содержать этот печатаемый символ. Для функциональных клавиш свойство `key` должно содержать такие значения, как «F2», «Home» или «Left».

На момент написания этих строк свойство `key`, определяемое стандартом «DOM Level 3 Events», еще не было реализовано ни в одном из броузеров. Однако броузеры на базе механизма Webkit, Safari и Chrome определяют в объектах этих событий свойство `keyIdentifier`. Для функциональных клавиш, подобно свойству `key`, свойство `keyIdentifier` содержит не число, а строку с именем клавиши, таким как «Shift» или «Enter». Для клавиш, соответствующих печатаемым символам, это свойство содержит менее удобное в использовании строковое представление кодового пункта Юникода символа. Например, клавише «A» соответствует значение «U+0041».

В примере 17.8 определяется класс `Keymap`, который отображает идентификаторы комбинаций клавиш, такие как «PageUp», «Alt_Z» и «ctrl+alt+shift+F5» в функции на языке JavaScript, вызываемые в ответ на нажатия этих комбинаций. Определения привязок клавиш передаются конструктору `Keymap()` в форме объекта JavaScript, имена свойств которого соответствуют идентификаторам комбинаций клавиш, а значения этих свойств содержат ссылки на функции-обработчики. Добавление и удаление привязок осуществляется с помощью методов `bind()` и `unbind()`. Устанавливается объект `Keymap` в HTML-элемент (обычно в объект `Document`) с помощью метода `install()`. При установке объекта `Keymap` в этом элементе регистрируется обработчик события «keydown». Каждый раз, когда нажимается клавиша, обработчик проверяет наличие функции, соответствующей этой комбинации. Если функция существует, она вызывается. Обработчик события «key-

down» использует свойство key, определяемое стандартом «DOM Level 3 Events», если оно существует. В противном случае он пытается использовать Webkit-свойство keyIdentifier. И как запасной вариант, обработчик использует нестандартное свойство keyCode. Пример 17.8 начинается с длинного комментария, подробно описывающего работу модуля.

Пример 17.8. Класс Keymap для обработки нажатий комбинаций клавиш

```
/*
 * Keymap.js: связывает события клавиатуры с функциями-обработчиками.
 *
 * Этот модуль определяет класс Keymap. Экземпляр этого класса представляет
 * собой отображение идентификаторов комбинаций клавиш (определяемых ниже)
 * в функции-обработчики. Объект Keymap можно установить в HTML-элемент
 * для обработки событий keydown. Когда возникает это событие, объект Keymap
 * использует свою карту привязок для вызова соответствующего обработчика.
 *
 * При создании объекта Keymap конструктору можно передать JavaScript-объект,
 * представляющий начальную карту привязок. Имена свойств этого объекта должны
 * соответствовать идентификаторам клавиш, а значениями должны быть функции-обработчики.
 * После создания объекта Keymap в него можно добавлять новые привязки, передавая
 * идентификатор клавиши и функцию-обработчик методу bind(). Имеется также возможность
 * удалить привязку, передав идентификатор клавиши методу unbind().
 *
 * Чтобы задействовать объект Keymap, следует вызвать его метод install(), передав ему
 * HTML-элемент, такой как объект document. Метод install() добавит в указанный объект
 * обработчик события onkeydown. Когда этот обработчик будет вызван, он определит
 * идентификатор нажатой клавиши и вызовет функцию-обработчик (если таковая имеется),
 * привязанную к этому идентификатору клавиши. Один и тот же объект Keymap
 * можно установить сразу в несколько HTML-элементов.
 *
 * Идентификаторы клавиш
 *
 * Идентификатор клавиши - это нечувствительная к регистру символов строка,
 * представляющая клавишу, плюс любое количество удерживаемых нажатыми
 * клавиш-модификаторов. Именем клавиши является основной текст, изображаемый
 * на клавише. Допустимыми именами клавиш являются: "A", "7", "F2", "PageUp",
 * "Left", "Backspace" и "Esc".
 *
 * Список имен находится в объекте Keymap.keyCodeToKeyName, внутри этого модуля.
 * Они являются подмножеством имен, определяемых стандартом "DOM Level 3".
 * Кроме того, этот класс будет использовать свойство key, когда оно будет реализовано.
 *
 * Идентификатор клавиши может также включать имена клавиш-модификаторов.
 * Это имена Alt, Ctrl, Meta и Shift. Они нечувствительны к регистру символов и должны
 * отделяться от имени клавиши и друг от друга пробелами или подчеркиваниями, дефисами
 * или знаками +. Например: "SHIFT+A", "Alt_F2", "meta-v" и "ctrl alt left".
 * В компьютерах Mac клавише Meta соответствует клавиша Command, а клавише Alt -
 * клавиша Option. Некоторые броузеры отображают клавишу Windows в клавишу Meta.
 *
 * Функции-обработчики
 *
 * Обработчики вызываются как методы объекта document или элемента документа,
 * в зависимости от того, куда был установлен объект Keymap, и им передаются
 * два аргумента:
 *   1) объект события keydown
 *   2) идентификатор нажатой клавиши
 * Значение, возвращаемое функцией, становится возвращаемым значением
 * обработчика события keydown. Если функция-обработчик вернет false,
 * объект Keymap прервет всплытие события и предотвратит выполнение любых
 * действий по умолчанию, связанных с событием keydown.
 *
 * Ограничения
 *
 * Функцию-обработчик можно привязать не ко всем клавишам. Некоторые комбинации
 * используются самой операционной системой (например, Alt-F4). А некоторые комбинации
 * могут перехватываться броузером (например, Ctrl-S). Эта реализация зависит
 * от особенностей броузера, ОС и региональных настроек. Вы с успехом можете
 * использовать функциональные клавиши и функциональные клавиши с модификаторами,
 * а также алфавитно-цифровые клавиши без модификаторов. Комбинации алфавитно-цифровых
 * клавиш с модификаторами Ctrl и Alt менее надежны.
 *
 * Поддерживается большинство знаков препинания, кроме дефиса, для ввода которых
 * не требуется удерживать клавишу Shift (`=[];',./\) на клавиатурах
 * со стандартной раскладкой US. Но они плохо совместимы с другими
 * раскладками клавиатур, и их желательно не использовать.
 */
// Функция-конструктор
function Keymap(bindings) {
    this.map = {};  // Определить отображение идентификатор->обработчик
    if (bindings) { // Скопировать в него начальную карту привязок
        for(name in bindings) this.bind(name, bindings[name]);
    }
}

// Связывает указанный идентификатор клавиши с указанной функцией-обработчиком
Keymap.prototype.bind = function(key, func) {
    this.map[Keymap.normalize(key)] = func;
};

// Удаляет привязку для указанного идентификатора клавиши
Keymap.prototype.unbind = function(key) {
    delete this.map[Keymap.normalize(key)];
};

// Устанавливает этот объект Keymap в указанный HTML-элемент
Keymap.prototype.install = function(element) {
    var keymap = this;
    // Определить функции-обработчика события
    function handler(event) { return keymap.dispatch(event, element); }

    // Установить ее
    if (element.addEventListener)
        element.addEventListener("keydown", handler, false);
    else if (element.attachEvent)
        element.attachEvent("onkeydown", handler);
};

// Этот метод делегирует обработку события клавиатуры, опираясь на привязки.
Keymap.prototype.dispatch = function(event, element) {
    // Изначально нет ни имен клавиш-модификаторов, ни имени клавиши
    var modifiers = ""
    var keyname = null;

    // Сконструировать строки модификаторов в каноническом виде из символов
    // в нижнем регистре, расположив их в алфавитном порядке.
    if (event.altKey) modifiers += "alt_";
    if (event.ctrlKey) modifiers += "ctrl_";
    if (event.metaKey) modifiers += "meta_";
    if (event.shiftKey) modifiers += "shift_";

    // Имя клавиши легко получить, если реализовано свойство key,
    // определяемое стандартом DOM Level 3:
    if (event.key) keyname = event.key;
    // Для получения имен функциональных клавиш в Safari и Chrome можно
    // использовать свойство keyIdentifier
    else if(event.keyIdentifier&&event.keyIdentifier.substring(0,2) !== "U+")
        keyname = event.keyIdentifier;
    // В противном случае можно использовать свойство keyCode и отображение код->имя ниже
    else keyname = Keymap.keyCodeToKeyName[event.keyCode];

    // Если имя клавиши не удалось определить, просто проигнорировать событие
    // и вернуть управление.
    if (!keyname) return;

    // Канонический идентификатор клавиши состоит из имен модификаторов
    // и имени клавиши в нижнем регистре
    var keyid = modifiers + keyname.toLowerCase();

    // Проверить, имеется ли привязка для данного идентификатора клавиши
    var handler = this.map[keyid];

    if (handler) { // Если обработчик для данной клавиши, вызвать его
        // Вызвать функцию-обработчик
        var retval = handler.call(element, event, keyid);

        // Если обработчик вернул false, отменить действия по умолчанию
        // и прервать всплытие события
        if (retval === false) {
            if (event.stopPropagation) event.stopPropagation(); // модель DOM
            else event.cancelBubble = true;                     // модель IE
            if (event.preventDefault) event.preventDefault();   // DOM
            else event.returnValue = false;                     // IE
        }

        // Вернуть значение, полученное от обработчика
        return retval;
    }
};

// Вспомогательная функция преобразования идентификатора клавиши в каноническую форму.
// Нам необходимо преобразовать идентификатор "meta" в "ctrl", чтобы превратить
// идентификатор Meta-C в "Command-C" на компьютерах Mac и в "Ctrl-C" на всех остальных.
Keymap.normalize = function(keyid) {
    keyid = keyid.toLowerCase();              // В нижний регистр
    var words = keyid.split(/\s+|[\-+_]/);    // Вычленить модификаторы
    var keyname = words.pop();                // keyname - последнее слово
    keyname = Keymap.aliases[keyname] || keyname; // Это псевдоним?
    words.sort();                                 // Сортировать модификаторы
    words.push(keyname);                          // Поместить обратно
                                                  // нормализованное имя
    return words.join("_");                       // Объединить все вместе
};

Keymap.aliases = {      // Отображение привычных псевдонимов клавиш в их
    "escape":"esc",     // "официальные" имена, используемые в DOM Level 3,
    "delete":"del",     // и отображение кодов клавиш в имена ниже.
    "return":"enter",   // Имя и значение должны состоять только из символов
    "ctrl":"control",   // нижнего регистра.
    "space":"spacebar",
    "ins":"insert"
};

// Старое свойство keyCode объекта события keydown не стандартизовано
// Но следующие значения с успехом могут использоваться в большинстве броузеров и ОС.
Keymap.keyCodeToKeyName = {
    // Клавиши со словами или стрелками на них
    8:"Backspace", 9:"Tab", 13:"Enter", 16:"Shift", 17:"Control", 18:"Alt",
    19:"Pause", 20:"CapsLock", 27:"Esc", 32:"Spacebar", 33:"PageUp",
    34:"PageDown", 35:"End", 36:"Home", 37:"Left", 38:"Up", 39:"Right",
    40:"Down", 45:"Insert", 46:"Del",

    // Цифровые клавиши на основной клавиатуре (не на дополнительной)
    48:"0",49:"1",50:"2",51:"3",52:"4",53:"5",54:"6",55:"7",56:"8",57:"9",

    // Буквенные клавиши. Обратите внимание, что здесь не различаются
    // символы верхнего и нижнего регистров
    65:"A", 66:"B", 67:"C", 68:"D", 69:"E", 70:"F", 71:"G", 72:"H", 73:"I",
    74:"J", 75:"K", 76:"L", 77:"M", 78:"N", 79:"O", 80:"P", 81:"Q", 82:"R",
    83:"S", 84:"T", 85:"U", 86:"V", 87:"W", 88:"X", 89:"Y", 90:"Z",

    // Цифровые клавиши на дополнительной клавиатуре и клавиши со знаками препинания.
    // (Не поддерживаются в Opera.)
    96:"0",97:"1",98:"2",99:"3",100:"4",101:"5",102:"6",103:"7",104:"8",
    105:"9",106:"Multiply", 107:"Add", 109:"Subtract", 110:"Decimal",
    111:"Divide",

    // Функциональные клавиши
    112:"F1", 113:"F2", 114:"F3", 115:"F4", 116:"F5", 117:"F6",
    118:"F7", 119:"F8", 120:"F9", 121:"F10", 122:"F11", 123:"F12",
    124:"F13", 125:"F14", 126:"F15", 127:"F16", 128:"F17", 129:"F18",
    130:"F19", 131:"F20", 132:"F21", 133:"F22", 134:"F23", 135:"F24",

    // Клавиши со знаками препинания, для ввода которых не требуется
    // удерживать нажатой клавишу Shift.
    // Дефис не может использоваться переносимым способом: FF возвращает
    // тот же код, что и для клавиши Subtract
    59:";", 61:"=", 186:";", 187:"=",     // Firefox и Opera возвращают 59,61
    188:",", 190:".", 191:"/", 192:"`", 219:"[", 220:"\\", 221:"]", 222:"'"
};
```

18

Работа с протоколом HTTP

Протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP) определяет, как веб-броузеры должны запрашивать документы, как они должны передавать информацию веб-серверам и как веб-серверы должны отвечать на эти запросы и передачи. Очевидно, что веб-броузеры очень много работают с протоколом HTTP. Тем не менее, как правило, сценарии не работают с протоколом HTTP, когда пользователь щелкает на ссылке, отправляет форму или вводит URL в адресной строке.

Однако JavaScript-код *способен* работать с протоколом HTTP. HTTP-запросы могут инициироваться, когда сценарий устанавливает значение свойства `location` объекта `Window` или вызывает метод `submit()` объекта `Form`. В обоих случаях броузер загружает в окно новую страницу. Такого рода взаимодействие с протоколом HTTP может быть вполне оправданным в веб-страницах, состоящих из нескольких фреймов, но в этой главе мы будем говорить совсем о другом. Здесь мы рассмотрим такое взаимодействие JavaScript-кода с веб-сервером, при котором веб-броузер не перезагружает содержимое окна или фрейма.

Термин *Ajax* описывает архитектуру веб-приложений, отличительной чертой которых является работа с протоколом HTTP.[1] Ключевой особенностью Ajax-приложения является использование протокола HTTP для инициации обмена данными с веб-сервером без необходимости перезагружать страницу. Возможность избежать перезагрузки страницы (что было привычным на первых этапах развития Всемирной паутины) позволяет создавать веб-приложения, близкие по своему поведению к обычным приложениям. Веб-приложение может использовать технологии Ajax для передачи на сервер результатов взаимодействия с пользователем или для ускорения запуска приложения, отображая сначала простую страницу и подгружая дополнительные данные и компоненты страницы по мере необходимости.

---

[1] Ajax – это аббревиатура от Asynchronous JavaScript and XML (асинхронный JavaScript и XML). Этот термин предложил Джесси Джеймс Гаррет (Jesse James Garrett) и впервые использовал его в своей статье «Ajax: A New Approach to Web Applications» в феврале 2005 года (*http://www.adaptivepath.com/publications/essays/archives/000385.php*) (От переводчика: перевод этой статьи на русский язык можно найти по адресу *http://galleo.ru/articles/w160.*). В течение многих лет термин «Ajax» был громким словом, которое употребляли к месту и не к месту, а сейчас это всего лишь удобный термин, обозначающий архитектуру веб-приложения, опирающегося в своей работе на HTTP-запросы.

Термин *Comet* описывает похожую архитектуру веб-приложений, также использующих протокол HTTP.[1] В некотором смысле архитектура Comet является обратной по отношению к Ajax: в архитектуре Comet не клиент, а сервер инициирует взаимодействие, асинхронно отсылая сообщения клиенту. Если веб-приложению потребуется отвечать на сообщения, отправляемые сервером, оно сможет использовать приемы Ajax для отправки или запроса данных. В архитектуре Ajax клиент «вытягивает» данные с сервера. В архитектуре Comet сервер «навязывает» данные клиенту. Иногда архитектуру Comet называют «Server Push», «Ajax Push» и «HTTP Streaming».

Есть множество способов реализации архитектур Ajax и Comet, и эти базовые реализации иногда называют *транспортами*. Элемент `<img>`, например, имеет свойство `src`. Когда сценарий записывает в это свойство URL-адрес, инициируется HTTP-запрос GET и выполняется загрузка содержимого с этого URL-адреса. Таким образом, сценарий может отправлять информацию веб-серверу, добавляя ее в виде строки запроса в URL-адрес изображения и устанавливая свойство `src` элемента `<img>`. В ответ на этот запрос веб-сервер должен вернуть некоторое изображение, которое, например, может быть невидимым: прозрачным и размером 1×1 пиксел.[2]

Элемент `<img>` – не самый лучший транспорт Ajax, потому что обмен данными ведется только в одном направлении: клиент может передать данные серверу, но ответом сервера всегда будет изображение, извлечь информацию из которого на стороне клиента очень непросто. Элемент `<iframe>` обладает большей гибкостью. При использовании элемента `<iframe>` в качестве транспорта Ajax сценарий сначала добавляет в URL-адрес информацию, предназначенную для веб-сервера, а затем записывает этот URL-адрес в свойство `src` тега `<iframe>`. Сервер создает HTML-документ, содержащий ответ на запрос, и отправляет его обратно веб-броузеру, который выводит ответ в теге `<iframe>`. При этом элемент `<iframe>` необязательно должен быть видимым для пользователя – он может быть сокрыт, например, средствами таблиц стилей CSS. Сценарий может проанализировать ответ сервера, выполнив обход документа в элементе `<iframe>`. Обратите внимание, что взаимодействие с документом ограничивается политикой общего происхождения, о которой рассказывается в разделе 13.6.2.

Даже изменение свойства `src` элемента `<script>` может использоваться для инициирования HTTP-запроса GET. Использование элементов `<script>` для работы с протоколом HTTP выглядит особенно привлекательно, потому что они не являются субъектами политики общего происхождения и могут использоваться для взаимодействий с разными серверами. Обычно при использовании транспорта Ajax на основе элемента `<script>` ответ сервера имеет вид данных в формате JSON (раздел 6.9), которые автоматически «декодируются», когда содержимое элемента `<script>` выполняется интерпретатором JavaScript. Из-за использования формата данных JSON этот транспорт Ajax получил название «JSONP».

---

[1] Имя Comet было предложено Алексом Расселом (Alex Russell) в статье «Comet: Low Latency Data for the Browser» (*http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/*). Вероятно, выбирая такое имя, Алекс Рассел хотел обыграть термин Ajax: дело в том, что в США Comet и Ajax являются названиями чистящих средств.

[2] Такие изображения иногда называют *веб-жучками* (*web bugs*). Они пользуются дурной славой из-за проблем с безопасностью, когда используются для обмена информацией со сторонним сервером, не тем, откуда была загружена страница. Одно из типичных применений веб-жучков – подсчет числа посещений и анализа трафика веб-сервера.

Хотя архитектура Ajax может быть реализована поверх транспорта `<iframe>` или `<script>`, существует более простой путь. Уже достаточно давно все броузеры стали поддерживать объект `XMLHttpRequest`, определяющий прикладной интерфейс для работы с протоколом HTTP. Этот интерфейс обеспечивает возможность выполнять POST-запросы в дополнение к обычным GET-запросам и может возвращать ответ веб-сервера синхронно или асинхронно, в виде простого текста или в виде объекта `Document`. Несмотря на свое название, объект `XMLHttpRequest` не ограничивается использованием XML-документов – он в состоянии принимать любые текстовые документы. Прикладной интерфейс объекта `XMLHttpRequest` рассматривается в разделе 18.1, который занимает значительную часть главы. Большая часть примеров реализации архитектуры Ajax в этой главе в качестве транспорта использует объект `XMLHttpRequest`, но в разделе 18.2 также будет показано, как использовать транспорт на основе элемента `<script>`, так как он способен обходить ограничения политики общего происхождения.

Транспортные механизмы в архитектуре Comet сложнее, чем в архитектуре Ajax, но все они требуют установления (и восстановления, в случае необходимости) соединения с сервером со стороны клиента и обязывают сервер поддерживать соединение открытым, чтобы через него можно было отправлять асинхронные сообщения. Транспортом в архитектуре Comet может служить, например, скрытый элемент `<iframe>`, если сервер отправляет сообщения в виде элементов `<script>`, выполняемых в элементе `<iframe>`. Более надежный и переносимый подход к реализации архитектуры Comet заключается в том, чтобы клиент устанавливал соединение с сервером (используя Ajax-транспорт), а сервер поддерживал это соединение открытым, пока имеется необходимость отправлять сообщения. Каждый раз, когда сервер отправляет сообщение, он закрывает соединение, что позволяет гарантировать благополучное получение сообщения клиентом. После обработки сообщения клиент может сразу же установить новое соединение для будущих сообщений.

Реализация надежного и переносимого транспорта для архитектуры Comet является сложной задачей, и большинство веб-разработчиков, использующих архитектуру Comet, опираются на транспорты, реализованные в веб-фреймворках, таких как Dojo. На момент написания этих строк производители броузеров приступили к реализации положений проекта спецификации «Server-Sent Events», связанной со стандартом HTML5, которая определяет простой прикладной интерфейс для архитектуры Comet в виде объекта `EventSource`. Интерфейс объекта `EventSource` будет рассматриваться в разделе 18.3 и там же будет продемонстрирована простая его имитация на основе объекта `XMLHttpRequest`.

Имеется также возможность конструировать высокоуровневые протоколы взаимодействий поверх Ajax и Comet. Эти приемы организации взаимодействий типа клиент/сервер можно использовать, например, в качестве основы механизма RPC (Remote Procedure Call – вызов удаленных процедур) или системы событий типа издатель/подписчик. Однако в данной главе мы не будем рассматривать подобные высокоуровневые протоколы, а сконцентрируемся на прикладных интерфейсах поддержки архитектур Ajax и Comet.

Использование XML не является обязательным

Символ «X» в аббревиатуре «Ajax» обозначает «XML». Основной прикладной интерфейс для работы с протоколом HTTP на стороне клиента (`XMLHttpRequest`) также содержит название «XML» в своем имени, и, как мы узнаем далее, одно из свойств объекта `XMLHttpRequest` имеет имя `responseXML`. Вследствие этого может сложиться впечатление, что XML является важной частью работы с протоколом HTTP. Но это не так: эти имена являются историческим наследием тех дней, когда XML было модным словечком. Конечно, реализации Ajax способны работать с XML-документами, но использование формата XML является необязательным, и в действительности он редко используется не практике. Спецификация XMLHttpRequest полна несоответствий в именах, с которыми нам придется сталкиваться:

> Имя объекта `XMLHttpRequest` было выбрано исходя из соображений совместимости с Веб, хотя каждая часть этого имени может вводить в заблуждение. Во-первых, объект поддерживает любые текстовые форматы, включая XML. Во-вторых, он может использоваться для отправки запросов по обоим протоколам, HTTP и HTTPS (некоторые реализации могут поддерживать дополнительные протоколы, помимо HTTP и HTTPS, но эти возможности не регламентируются в спецификации). Наконец, он поддерживает «запросы» («requests») в более широком смысле, – как это подразумевает использование протокола HTTP. А именно, все операции, связанные с выполнением HTTP-запросов или получением HTTP-ответов для определенных HTTP-методов.

18.1. Использование объекта XMLHttpRequest

Прикладной интерфейс к протоколу HTTP в броузерах определяется в виде класса `XMLHttpRequest`. Каждый экземпляр этого класса представляет единственную пару запрос/ответ, а свойства и методы объекта позволяют определять параметры запроса и извлекать данные из ответа. Объект `XMLHttpRequest` поддерживается веб-броузерами уже довольно давно, а его прикладной интерфейс находится на последних стадиях стандартизации консорциумом W3C. В то же время в консорциуме W3C ведутся работы над проектом стандарта «XMLHttpRequest Level 2». В этом разделе мы рассмотрим базовый прикладной интерфейс объекта `XMLHttpRequest`, а также те части проекта стандарта «XMLHttpRequest Level 2» (я называю его «XHR2»), которые в настоящее время реализованы как минимум в двух броузерах.

Первое, что обычно необходимо сделать при использовании этого прикладного интерфейса к протоколу HTTP, это, разумеется, создать экземпляр объекта `XMLHttpRequest`:

```
var request = new XMLHttpRequest();
```

Допустимо повторно использовать уже имеющийся экземпляр объекта `XMLHttpRequest`, но следует иметь в виду, что в этом случае будет прервано выполнение запроса, уже отправленного объектом.

XMLHttpRequest в IE6

Корпорация включила поддержку объекта `XMLHttpRequest` в свой броузер IE, начиная с версии 5. В версиях IE5 и IE6 этот объект доступен только в виде ActiveX-объекта. Поддержка современного стандартного конструктора `XMLHttpRequest()` появилась только в IE7, но его можно имитировать, как показано ниже:

```
// Имитация конструктора XMLHttpRequest() в IE5 и IE6
if (window.XMLHttpRequest === undefined) {
    window.XMLHttpRequest = function() {
        try {
            // Использовать последнюю версию ActiveX-объекта, если доступна
            return new ActiveXObject("Msxml2.XMLHTTP.6.0");
        }
        catch (e1) {
            try {
                // Иначе вернуться к старой версии
                return new ActiveXObject("Msxml2.XMLHTTP.3.0");
            }
            catch(e2) {
                // Если ничего не получилось - возбудить ошибку
                throw new Error("XMLHttpRequest не поддерживается");
            }
        }
    };
}
```

HTTP-запрос состоит из четырех частей:

- метод HTTP-запроса или тип «операции»
- запрашиваемый URL-адрес
- необязательные заголовки запроса, которые могут включать информацию для аутентификации
- необязательное тело запроса

HTTP-ответ, возвращаемый сервером, состоит из трех частей:

- числовое и текстовое значение, определяющее код состояния, свидетельствующий об успехе или об ошибке
- набор заголовков ответа
- тело ответа

Первые два подраздела, следующие далее, демонстрируют, как устанавливать каждую часть HTTP-запроса и как извлекать части из HTTP-ответа. За этими ключевыми разделами следуют подразделы, освещающие более узкоспециализированные темы.

Базовая архитектура запрос/ответ протокола HTTP весьма проста в использовании. Однако на практике возникает масса сложностей: клиенты и серверы обмениваются данными в виде cookies; серверы переадресуют броузеры на другие серверы; одни ресурсы кэшируются, а другие – нет; некоторые клиенты отправляют запросы через прокси-серверы и т. д. Объект `XMLHttpRequest` не является прикладным интерфейсом уровня протокола, он обеспечивает прикладной интерфейс уровня броузера. Броузер сам заботится о cookies, переадресации, кэшировании и прокси-серверах, а вам достаточно позаботиться только о запросах и ответах.

XMLHttpRequest и локальные файлы

Возможность использования относительных URL-адресов в веб-страницах обычно означает, что HTML-страницы можно разрабатывать и проверять, используя локальную файловую систему, а затем перемещать их на веб-сервер без дополнительных изменений. Однако, как правило, это невозможно при использовании архитектуры Ajax на основе объекта `XMLHttpRequest`. Объект `XMLHttpRequest` предназначен для работы с протоколами HTTP и HTTPS. Теоретически он мог было работать с другими протоколами, такими как FTP, но такие части прикладного интерфейса, как метод запроса и код состояния ответа, являются характерными именно для протокола HTTP. Если загрузить веб-страницу из локального файла, сценарии в этой странице не смогут использовать объект `XMLHttpRequest` с относительными URL-адресами, потому что эти адреса будут относительными адресами вида `file://`, а не `http://`. А политика общего происхождения зачастую будет препятствовать использованию абсолютных адресов вида `http://`. (Тем не менее загляните в раздел 18.1.6.) Таким образом, чтобы проверить веб-страницы, использующие объект `XMLHttpRequest`, их необходимо выгружать на веб-сервер (или использовать локальный веб-сервер).

18.1.1. Выполнение запроса

Следующий этап после создания объекта `XMLHttpRequest` – определение параметров HTTP-запроса вызовом метода `open()` объекта `XMLHttpRequest`, которому передаются две обязательные части запроса: метод и URL:

```
request.open("GET",          // Запрос типа HTTP GET
             "data.csv");  // на получение содержимого по этому URL-адресу
```

Первый аргумент метода `open()` определяет HTTP-метод или операцию. Это строка, не чувствительная к регистру, но обычно содержащая только символы верхнего регистра, в соответствии со спецификацией протокола HTTP. Методы «GET» и «POST» поддерживаются всеми броузерами. Метод «GET» используется для «обычных» запросов и соответствует случаю, когда URL-адрес полностью определяет запрашиваемый ресурс. Он используется, когда запрос не имеет побочных эффектов и когда ответ сервера можно поместить в кэш. Метод «POST» обычно используется HTML-формами. Он включает в тело запроса дополнительные данные (данные формы), и эти данные часто сохраняются в базе данных на стороне сервера (побочный эффект). В ответ на повторяющиеся запросы POST к одному и тому же URL сервер может возвращать разные ответы, и ответы на запросы, отправленные этим методом, не должны помещаться в кэш.

Помимо запросов «GET» и «POST», спецификация XMLHttpRequest также позволяет передавать методу `open()` строки «DELETE», «HEAD», «OPTIONS» и «PUT» в первом аргументе. (Методы «HTTP CONNECT», «TRACE» и «TRACK» явно запрещены к использованию из-за проблем с безопасностью.) Старые версии броузеров могут не поддерживать все эти методы, но метод «HEAD», по крайней мере, поддерживается почти всеми броузерами, и его использование демонстрируется в примере 18.13.

Вторым аргументом методу `open()` передается URL-адрес запрашиваемого ресурса. Это относительный URL-адрес документа, содержащего сценарий, в котором вызывается метод `open()`. Если указать абсолютный адрес, то в общем случае протокол, доменное имя и порт должны совпадать с аналогичными параметрами адреса документа: нарушение политики общего происхождения обычно вызывает ошибку. (Однако спецификация «XMLHttpRequest Level 2» допускает выполнение запросов к другим серверам, если сервер явно разрешил это – смотрите раздел 18.1.6.)

Следующий этап в выполнении запроса – установка заголовков запроса, если это необходимо. Запросы POST, например, требуют, чтобы был определен заголовок «Content-Type», определяющий MIME-тип тела запроса:

```
request.setRequestHeader("Content-Type", "text/plain");
```

Если вызвать метод `setRequestHeader()` несколько раз с одним и тем же заголовком, новое значение не заменит прежнее: вместо этого в HTTP-запрос будет вставлено несколько копий заголовка или один заголовок с несколькими значениями.

Нельзя определять собственные заголовки «Content-Length», «Date», «Referer» и «User-Agent»: объект `XMLHttpRequest` добавляет их автоматически и не позволяет подделывать их. Аналогично объект `XMLHttpRequest` автоматически обрабатывает cookies и срок поддержки открытого соединения, определяет кодировку символов и выполняет кодирование сообщений, поэтому вы не должны передавать методу `setRequestHeader()` следующие заголовки:

```
Accept-Charset      Content-Transfer-Encoding   TE
Accept-Encoding     Date                        Trailer
Connection          Expect                      Transfer-Encoding
Content-Length      Host                        Upgrade
Cookie              Keep-Alive                  User-Agent
Cookie2             Referer                     Via
```

В запросе можно определить заголовок «Authorization», но обычно в этом нет необходимости. При выполнении запроса к ресурсу, защищенному паролем, передайте имя пользователя и пароль методу `open()` в четвертом и пятом аргументах, а объект `XMLHttpRequest` автоматически установит соответствующие заголовки. (О третьем необязательном аргументе метода `open()` рассказывается ниже, а описание аргументов, в которых передаются имя пользователя и пароль, можно найти в справочной части книги.)

Последний этап в процедуре выполнения HTTP-запроса с помощью объекта `XMLHttpRequest` – передача необязательного тела запроса и отправка его серверу. Делается это с помощью метода `send()`:

```
request.send(null);
```

GET-запросы не имеют тела, и в этом случае можно передать методу значение `null` или вообще опустить аргумент. POST-запросы обычно имеют тело, и оно должно соответствовать заголовку «Content-Type», установленному с помощью метода `setRequestHeader()`.

Пример 18.1 демонстрирует использование всех методов объекта `XMLHttpRequest`, описанных выше. Он отправляет серверу текстовую строку методом POST и игнорирует ответ, возвращаемый сервером.

Пример 18.1. Отправка простого текста на сервер методом POST

```
function postMessage(msg) {
    var request = new XMLHttpRequest();      // Новый запрос
    request.open("POST", "/log.php");        // серверному сценарию методом POST
    // Отправить простое текстовое сообщение в теле запроса
    request.setRequestHeader("Content-Type", // Тело запроса - простой текст
                         "text/plain;charset=UTF-8");
    request.send(msg);                       // msg как тело запроса
    // Запрос выполнен. Мы игнорируем возможный ответ или ошибку.
}
```

Обратите внимание, что вызов метода `send()` в примере 18.1 инициирует запрос и затем возвращает управление: он не блокируется в ожидании ответа от сервера. HTTP-ответы практически всегда обрабатываются асинхронно, как будет показано в следующем разделе.

Порядок имеет значение

Части HTTP-запроса следуют в определенном порядке: метод запроса и URL-адрес должны определяться в первую очередь, затем должны устанавливаться заголовки запроса и, наконец, тело запроса. Обычно реализации `XMLHttpRequest` ничего не отправляют в сеть, пока не будет вызван метод `send()`. Но прикладной интерфейс `XMLHttpRequest` спроектирован так, как если бы каждый метод немедленно отправлял данные в сеть. Это означает, что методы объекта `XMLHttpRequest` должны вызываться в порядке, соответствующем структуре HTTP-запроса. Например, метод `setRequestHeader()` должен вызываться после метода `open()` и перед методом `send()`, в противном случае он возбудит исключение.

18.1.2. Получение ответа

Полный HTTP-ответ содержит код состояния, набор заголовков ответа и тело ответа. Все это доступно в виде свойств и методов объекта `XMLHttpRequest`:

- Свойства `status` и `statusText` возвращают код состояния HTTP в числовом и текстовом виде. Эти свойства хранят стандартные HTTP-значения, такие как 200 и «OK» в случае успешного выполнения запроса или 404 и «Not Found» при попытке обратиться к ресурсу, отсутствующему на сервере.
- Заголовки ответа можно получить с помощью методов `getResponseHeader()` и `getAllResponseHeaders()`. Обработка cookies выполняется объектом `XMLHttpRequest` автоматически: он исключает заголовки «Cookie» из множества, возвращае-

мого методом `getAllResponseHeaders()`, и возвращает `null`, если передать аргумент «Set-Cookie» или «Set-Cookie2» методу `getResponseHeader()`.

- Тело ответа в текстовом виде доступно через свойство `responseText` или в виде объекта `Document` через свойство `responseXML`. (Выбор такого имени свойства объясняется историческими причинами: фактически оно предназначено для работы с XHTML- и XML-документами, но спецификация «XHR2» определяет, что оно также должно работать с обычными HTML-документами.) Более подробно о свойстве `responseXML` рассказывается в разделе 18.1.2.2.

Обычно объект `XMLHttpRequest` используется в асинхронном режиме (но загляните в раздел 18.1.2.1): метод `send()` возвращает управление сразу же после отправки запроса, поэтому методы и свойства, перечисленные выше, не могут использоваться до фактического получения ответа. Чтобы определить момент получения ответа, необходимо обрабатывать событие «readystatechange» (или событие «progress», определяемое новой спецификацией «XHR2» и описываемое в разделе 18.1.4), возбуждаемое в объекте `XMLHttpRequest`. Но, чтобы понять, как обрабатывать это событие, необходимо сначала разобраться со свойством `readyState`.

Свойство `readyState` – это целочисленное значение, определяющее код состояния HTTP-запроса; его возможные значения перечислены в табл. 18.1. Идентификаторы, указанные в первой колонке, – это константы, определяемые конструктором `XMLHttpRequest`. Эти константы являются частью спецификации `XMLHttpRequest`, но старые версии броузеров и IE8 не определяют их, поэтому часто можно увидеть программный код, в котором вместо константы `XMLHttpRequest.DONE` используется числовое значение 4.

Теоретически событие «readystatechange» генерируется всякий раз, когда изменяется значение свойства `readyState`. На практике же событие может не возбуждаться, когда свойство `readyState` получает значение 0 или 1. Оно часто возбуждается при вызове метода `send()`, даже при том, что свойство `readyState` по-прежнему содержит значение `OPENED`. Некоторые броузеры возбуждают событие множество раз для состояния `LOADING`, чтобы обеспечить обратную связь. Все броузеры возбуждают событие «readystatechange», когда завершается прием ответа сервера и свойство `readyState` получает значение 4. Так как это событие может возбуждаться еще до завершения приема ответа, обработчики события «readystatechange» всегда должны проверять значение свойства `readyState`.

Чтобы обрабатывать события «readystatechange», нужно присвоить функцию обработчика события свойству `onreadystatechange` объекта `XMLHttpRequest`. Можно также воспользоваться методом `addEventListener()` (или `attachEvent()` в IE версии 8 и ниже), но обычно для обработки запроса бывает вполне достаточно одного обработчика, поэтому проще установить свойство `onreadystatechange`.

Таблица 18.1. Значения свойства readyState объекта XMLHttpRequest

Константа	Значение	Смысл
`UNSENT`	0	Метод `open()` еще не был вызван
`OPENED`	1	Метод `open()` был вызван
`HEADERS_RECEIVED`	2	Были получены заголовки
`LOADING`	3	Идет прием тела ответа
`DONE`	4	Прием ответа завершен

Пример 18.2 определяет функцию `getText()`, которая демонстрирует особенности обработки событий «readystatechange». Обработчик события сначала проверяет завершение запроса. После этого он проверяет код состояния ответа и убеждается в успешном выполнении. Затем он извлекает заголовок «Content-Type», чтобы убедиться, что получен ответ ожидаемого типа. Если выполняются все три условия, он передает тело ответа (в виде текста) указанной функции обратного вызова.

Пример 18.2. Получение HTTP-ответа в обработчике onreadystatechange

```
// Выполняет запрос HTTP GET содержимого указанного URL-адреса.
// После успешного получения ответа проверяет, содержит ли он простой текст,
// и передает его указанной функции обратного вызова
function getText(url, callback) {
    var request = new XMLHttpRequest();          // Создать новый запрос
    request.open("GET", url);                    // Указать URL-адрес ресурса
    request.onreadystatechange = function() {    // Определить обработчик события
        // Если запрос был выполнен успешно
        if (request.readyState === 4 && request.status === 200) {
            var type = request.getResponseHeader("Content-Type");
            if (type.match(/^text/))             // Убедиться, что это текст
                callback(request.responseText); // Передать функции
        }
    };
    request.send(null);                          // Отправить запрос
}
```

18.1.2.1. Получение синхронного ответа

Сама природа HTTP-ответа предполагает их асинхронную обработку. Тем не менее объект `XMLHttpRequest` поддерживает возможность получения ответов в синхронном режиме. Если в третьем аргументе передать методу `open()` значение `false`, выполнение метода `send()` будет заблокировано до завершения запроса. В этом случае отпадает необходимость использовать обработчик события: после того как метод `send()` вернет управление, можно будет сразу же проверить свойства `status` и `responseText` объекта `XMLHttpRequest`. Сравните следующую синхронную реализацию функции `getText()` из примера 18.2:

```
// Выполняет синхронный запрос HTTP GET содержимого по указанному URL-адресу.
// Возвращает текст ответа. Возбуждает исключение в случае неудачи
// или если ответ не является текстом.
function getTextSync(url) {
    var request = new XMLHttpRequest(); // Создать новый запрос
    request.open("GET", url, false);    // false - синхронный режим
    request.send(null);                 // Отправить запрос

    // Возбудить исключение, если код состояния не равен 200
    if (request.status !== 200) throw new Error(request.statusText);

    // Возбудить исключение, если ответ имеет недопустимый тип
    var type = request.getResponseHeader("Content-Type");
    if (!type.match(/^text/))
        throw new Error("Ожидался текстовый ответ; получен: " + type);

    return request.responseText;
}
```

Синхронные запросы выглядят весьма заманчиво, однако использовать их нежелательно. Интерпретатор JavaScript на стороне клиента выполняется в единственном потоке, и когда метод `send()` блокируется, это обычно приводит к зависанию пользовательского интерфейса всего броузера. Если сервер, к которому выполнено подключение, отвечает на вопросы с задержкой, броузер пользователя будет зависать. Тем не менее в разделе 22.4 вы познакомитесь с одним из случаев, когда синхронные запросы вполне допустимы.

18.1.2.2. Декодирование ответа

В примерах выше предполагалось, что сервер возвращает ответ в виде простого текста, с MIME-типом «text/plain», «text/html» или «text/css», и мы извлекаем его из свойства `responseText` объекта `XMLHttpRequest`.

Однако существуют и другие способы обработки ответов сервера. Если сервер посылает в ответе XML- или XHTML-документ, разобранное представление XML-документа можно получить из свойства `responseXML`. Значением этого свойства является объект `Document`, при работе с которым можно использовать приемы, представленные в главе 15. (Проект спецификации «XHR2» требует, чтобы броузеры автоматически выполняли синтаксический анализ ответов типа «text/html» и также делали их доступными через свойство `responseXML` в виде объектов `Document`, но на момент написания этих строк это требование не было реализовано в текущих броузерах.)

Если в ответ на запрос серверу потребуется отправить структурированные данные, такие как объект или массив, он может передать данные в виде строки в формате JSON (раздел 6.9). После получения такой строки содержимое свойства `responseText` можно передать методу `JSON.parse()`. Пример 18.3 является обобщенной версией примера 18.2: он выполняет запрос методом GET по указанному URL-адресу и после получения содержимого этого адреса передает его указанной функции обратного вызова. Но теперь функции не всегда будет передаваться простой текст – ей может быть передан объект `Document`, объект, полученный с помощью `JSON.parse()`, или строка.

Пример 18.3. Синтаксический анализ HTTP-ответа

```
// Выполняет запрос HTTP GET на получение содержимого по указанному URL-адресу.
// При получении ответа он передается функции обратного вызова
// как разобранный объект XML-документа, объект JSON или строка.
function get(url, callback) {
    var request = new XMLHttpRequest();       // Создать новый запрос
    request.open("GET", url);                 // Указать URL-адрес ресурса
    request.onreadystatechange = function() { // Определить обработчик события
        // Если запрос был выполнен и увенчался успехом
        if (request.readyState === 4 && request.status === 200) {
            // Определить тип ответа
            var type = request.getResponseHeader("Content-Type");
            // Проверить тип, чтобы избежать в будущем передачи ответа
            // в виде документа в формате HTML
            if (type.indexOf("xml") !== -1 && request.responseXML)
                callback(request.responseXML);             // Объект XML
            else if (type === "application/json")
                callback(JSON.parse(request.responseText)); // Объект JSON
            else
```

```
                callback(request.responseText);            // Строка
            }
        };
        request.send(null); // Отправить запрос
    }
```

Функция в примере 18.3 проверяет заголовок «Content-Type» ответа и обрабатывает ответы типа «application/json» особым образом. Другими типами ответов, которые может потребоваться «декодировать» особо, являются «application/JavaScript» и «text/JavaScript». С помощью объекта `XMLHttpRequest` можно запрашивать сценарии на языке JavaScript и затем выполнять их с помощью глобальной функции `eval()` (раздел 4.12.2). Однако в этом нет никакой необходимости, потому что возможностей самого элемента `<script>` вполне достаточно, чтобы загрузить и выполнить сценарий. Вернитесь к примеру 13.4, держа в уме, что элемент `<script>` может выполнять HTTP-запросы к другим серверам, запрещенные в прикладном интерфейсе `XMLHttpRequest`.

В ответ на HTTP-запросы веб-серверы часто возвращают двоичные данные (например, файлы изображений). Свойство `responseText` предназначено только для текстовых данных и не позволяет корректно обрабатывать ответы с двоичными данными, даже если использовать метод `charCodeAt()` полученной строки. Спецификация «XHR2» определяет способ обработки ответов с двоичными данными, но на момент написания этих строк производители броузеров еще не реализовали его. Дополнительные подробности приводятся в разделе 22.6.2.

Для корректного декодирования ответа сервера необходимо, чтобы сервер отправлял заголовок «Content-Type» с правильным значением MIME-типа ответа. Если, к примеру, сервер отправит XML-документ, не указав соответствующий MIME-тип, объект `XMLHttpRequest` не произведет синтаксический анализ ответа и не установит значение свойства `responseXML`. Или, если сервер укажет неправильное значение в параметре «charset» заголовка «Content-Type», объект `XMLHttpRequest` декодирует ответ с использованием неправильной кодировки и в свойстве `responseText` могут оказаться ошибочные символы.

Спецификация «XHR2» определяет метод `overrideMimeType()`, предназначенный для решения этой проблемы, и он уже реализован в некоторых броузерах. Если необходимо определить MIME-тип, лучше подходящий для веб-приложения, чем возвращаемый сервером, можно перед вызовом метода `send()` передать методу `overrideMimeType()` свой тип – это заставит объект `XMLHttpRequest` проигнорировать заголовок «Content-Type» и использовать указанный тип. Предположим, что необходимо загрузить XML-файл, который планируется интерпретировать как простой текст. В этом случае можно воспользоваться методом `overrideMimeType()`, чтобы сообщить объекту `XMLHttpRequest`, что он не должен выполнять синтаксический анализ файла и преобразовывать его в объект XML-документа:

```
// Не обрабатывать ответ, как XML-документ
request.overrideMimeType("text/plain; charset=utf-8")
```

18.1.3. Оформление тела запроса

Запросы HTTP POST включают тело запроса, которое содержит данные, передаваемые клиентом серверу. В примере 18.1 тело запроса было простой текстовой строкой. Однако нередко бывает необходимо передать в HTTP-запросе более сложные данные. В этом разделе демонстрируются некоторые способы реализации отправки таких данных.

18.1.3.1. Запросы с данными в формате HTML-форм

Рассмотрим HTML-формы. Когда пользователь отправляет форму, данные в форме (имена и значения всех элементов формы) помещаются в строку и отправляются вместе с запросом. По умолчанию HTML-формы отправляются на сервер методом POST, и данные формы помещаются в тело запроса. Схема преобразования данных формы в строку относительно проста: к имени и значению каждого элемента формы применяется обычное URI-кодирование (замена специальных символов шестнадцатеричными кодами), кодированные представления имен и значений отделяются знаком равенства, а пары имя/значение – амперсандом. Представление простой формы в виде строки может выглядеть, как показано ниже:

```
find=pizza&zipcode=02134&radius=1km
```

Такой формат представления данных формы соответствует формальному MIME-типу:

```
application/x-www-form-urlencoded
```

Этот тип следует указать в заголовке «Content-Type» запроса при отправке данных такого вида в составе запроса метолом POST.

Обратите внимание, что для использования такого формата представления не требуется наличие HTML-формы, и в действительности в этой главе мы не будем работать с формами непосредственно. В Ajax-приложениях чаще всего у вас будет иметься некоторый JavaScript-объект, который необходимо отправить на сервер. (Этот объект может быть создан из полей ввода HTML-формы, но в данном случае это не имеет значения.) Данные, показанные выше, могут оказаться представлением следующего JavaScript-объекта:

```
{
    find: "pizza",
    zipcode: 02134,
    radius: "1km"
}
```

Формат представления данных форм настолько широко используется во Всемирной паутине и настолько хорошо поддерживается всеми языками программирования для создания серверных сценариев, что его применение для представления данных, никак не связанных с формами, часто оказывается наиболее простым и удобным. Пример 18.4 демонстрирует, как преобразовать свойства объекта в формат представления формы.

Пример 18.4. Преобразование объекта в формат для отправки в HTTP-запросе

```
/**
 * Представляет свойства объекта, как если бы они были парами имя/значение
 * HTML-формы, с использованием формата application/x-www-form-urlencoded
 */
function encodeFormData(data) {
    if (!data) return "";                     // Всегда возвращать строку
    var pairs = [];                           // Для пар имя/значение
    for(var name in data) {                   // Для каждого имени
```

```
        if (!data.hasOwnProperty(name)) continue;      // Пропустить унаслед.
        if (typeof data[name] === "function") continue;// Пропустить методы
        var value = data[name].toString();             // Знач. в виде строки
        name = encodeURIComponent(name.replace("%20", "+"));   // Кодировать имя
        value = encodeURIComponent(value.replace("%20", "+")); // Кодировать значение
        pairs.push(name + "=" + value);  // Сохранить пару имя/значение
    }
    return pairs.join('&');              // Объединить пары знаком & и вернуть
}
```

С помощью этой функции `encodeFormData()` легко можно создавать утилиты, такие как функция `postData()`, представленная в примере 18.5. Обратите внимание, что для простоты эта функция `postData()` (и аналогичные ей функции в примерах ниже) не обрабатывает ответ сервера. При получении ответа она передает объект `XMLHttpRequest` целиком указанной функции обратного вызова. Эта функция обратного вызова сама должна проверять код состояния ответа и извлекать содержимое ответа.

Пример 18.5. Выполнение запроса HTTP POST с данными в формате представления форм

```
function postData(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);                     // Методом POST на указ. url
    request.onreadystatechange = function() {      // Простой обработчик
        if (request.readyState === 4 && callback) // При получении ответа
            callback(request);                     // вызвать указанную функцию
    };
    request.setRequestHeader("Content-Type",       // Установить "Content-Type"
                             "application/x-www-form-urlencoded");
    request.send(encodeFormData(data));            // Отправить данные
}                                                  // в представлении форм
```

Данные формы также могут быть отправлены посредством GET-запроса, и когда цель формы состоит в том, чтобы определить параметры операции чтения, метод GET лучше соответствует назначению формы, чем метод POST. GET-запросы не имеют тела, поэтому «полезный груз» с данными формы отправляется серверу в виде строки запроса в адресе URL (следующей за знаком вопроса). Утилита `encodeFormData()` может также пригодиться для отправки подобных GET-запросов, и пример 18.6 демонстрирует такое ее применение.

Пример 18.6. Выполнение GET-запроса с данными в формате представления форм

```
function getData(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("GET", url +                 // Методом GET на указанный url
                 "?" + encodeFormData(data)); // с добавлением данных
    request.onreadystatechange = function() { // Простой обработчик событий
        if (request.readyState === 4 && callback) callback(request);
    };
    request.send(null);                       // Отправить запрос
}
```

Для добавления данных в URL-адреса HTML-формы используют строки запросов, но использование объекта `XMLHttpRequest` дает свободу представления данных.

При наличии соответствующей поддержки на сервере наш запрос на поиск пиццерии можно оформить в виде более удобочитаемого URL-адреса, такого как показано ниже:

```
http://restaurantfinder.example.com/02134/1km/pizza
```

18.1.3.2. Запросы с данными в формате JSON

Использование формата представления данных форм в теле POST-запросов является распространенным соглашением, но не является обязательным требованием протокола HTTP. В последние годы в роли формата обмена данными во Всемирной паутине все большей популярностью пользуется формат JSON. Пример 18.7 демонстрирует, как с помощью функции `JSON.stringify()` (раздел 6.9) можно сформировать тело запроса. Обратите внимание, что этот пример отличается от примера 18.5 только последними двумя строками.

Пример 18.7. Выполнение запроса HTTP POST с данными в формате JSON

```
function postJSON(url, data, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);                    // Методом POST на указ. url
    request.onreadystatechange = function() {     // Простой обработчик
        if (request.readyState === 4 && callback) // При получении ответа
            callback(request);                    // вызвать указанную функцию
    };
    request.setRequestHeader("Content-Type", "application/json");
    request.send(JSON.stringify(data));
}
```

18.1.3.3. Запросы с данными в формате XML

Иногда для представления передаваемых данных также используется формат XML. Данные в запросе информации о пиццерии можно было бы передавать не в формате представления данных форм и не в формате JSON представления JavaScript-объектов, а в формате XML-документа. Тело такого запроса могло бы иметь следующий вид:

```
<query>
  <find zipcode="02134" radius="1km">
    pizza
  </find>
</query>
```

Во всех примерах, встречавшихся до сих пор, аргументом метода `send()` объекта `XMLHttpRequest` была строка или значение `null`. В действительности же этому методу можно также передать объект `Document` XML-документа. Пример 18.8 демонстрирует, как создать объект `Document` простого XML-документа и использовать его в качестве тела HTTP-запроса.

Пример 18.8. Выполнение запроса HTTP POST с XML-документом в качестве тела

```
// Параметры поиска "что", "где" и "радиус" оформляются в виде XML-документа
// и отправляются по указанному URL-адресу. При получении ответа вызывает
// указанную функцию
function postQuery(url, what, where, radius, callback) {
    var request = new XMLHttpRequest();
    request.open("POST", url);              // Методом POST на указанный url
    request.onreadystatechange = function() { // Простой обработчик
        if (request.readyState === 4 && callback) callback(request);
    };

    // Создать XML-документ с корневым элементом <query>
    var doc = document.implementation.createDocument("", "query", null);
    var query = doc.documentElement;          // Элемент <query>
    var find = doc.createElement("find");     // Создать элемент <find>
    query.appendChild(find);                  // И добавить в <query>
    find.setAttribute("zipcode", where);      // Атрибуты <find>
    find.setAttribute("radius", radius);
    find.appendChild(doc.createTextNode(what)); // И содержимое <find>

    // Отправить данные в формате XML серверу.
    // Обратите внимание, что заголовок Content-Type будет установлен автоматически.
    request.send(doc);
}
```

Обратите внимание, что пример 18.8 не устанавливает заголовок «Content-Type» запроса. Когда методу `send()` передается XML-документ, то объект `XMLHttpRequest` автоматически установит соответствующий заголовок «Content-Type», если он не был установлен предварительно. Аналогично, если передать методу `send()` простую строку и не установить заголовок «Content-Type», объект `XMLHttpRequest` автоматически добавит этот заголовок со значением «text/plain; charset=UTF-8». Программный код в примере 18.1 явно устанавливает этот заголовок, но в действительности для данных в простом текстовом виде этого не требуется.

18.1.3.4. Выгрузка файлов

Одна из особенностей HTML-форм заключается в том, что, если пользователь выберет файл с помощью элемента `<input type="file">`, форма отправит содержимое этого файла в теле POST-запроса. HTML-формы всегда позволяли выгружать файлы, но до недавнего времени эта операция была недоступна в прикладном интерфейсе `XMLHttpRequest`. Прикладной интерфейс, определяемый спецификацией «XHR2», позволяет выгружать файлы за счет передачи объекта `File` методу `send()`.

В данном случае нельзя создать объект с помощью конструктора `File()`: сценарий может лишь получить объект `File`, представляющий файл, выбранный пользователем. В броузерах, поддерживающих объекты `File`, каждый элемент `<input type="file">` имеет свойство `files`, которое ссылается на объект, подобный массиву, хранящий объекты `File`. Прикладной интерфейс буксировки (drag-and-drop) (раздел 17.7) также позволяет получить доступ к файлам, «сбрасываемым» пользователем на элемент, через свойство `dataTransfer.files` события «drop». Поближе с объектом `File` мы познакомимся в разделах 22.6 и 22.7. А пока будем рассматривать объект `File` как полностью непрозрачное представление выбранного пользователем файла, пригодное для выгрузки методом `send()`. В примере 18.9 представлена ненавязчивая JavaScript-функция, добавляющая обработчик события «change» к указанным элементам выгрузки файлов, чтобы они автоматически отправляли содержимое любого выбранного файла методом POST на указанный адрес URL.

Пример 18.9. Выгрузка файла посредством запроса HTTP POST

```
// Отыскивает все элементы <input type="file"> с атрибутом data-uploadto
// и регистрирует обработчик onchange, который автоматически отправляет
// выбранный файл по указанному URL-адресу "uploadto". Ответ сервера игнорируется.
whenReady(function() {                    // Вызвать эту функцию после загрузки документа
    var elts = document.getElementsByTagName("input"); // Все элементы input
    for(var i = 0; i < elts.length; i++) {          // Обойти в цикле
        var input = elts[i];
        if (input.type !== "file") continue;        // Пропустить все, кроме
                                                    // элементов выгрузки файлов
        var url = input.getAttribute("data-uploadto"); // Адрес выгрузки
        if (!url) continue;                         // Пропустить элементы без url

        input.addEventListener("change", function() { // При выборе файла
            var file = this.files[0];    // Предполагается выбор единственного файла
            if (!file) return;           // Если файл не выбран, ничего не делать
            var xhr = new XMLHttpRequest(); // Создать новый запрос
            xhr.open("POST", url);          // Методом POST на указанный URL
            xhr.send(file);                 // Отправить файл в теле запроса
        }, false);
    }
});
```

Как будет показано в разделе 22.6, тип `File` **является подтипом более общего типа** `Blob`**. Спецификация «XHR2» позволяет передавать методу** `send()` **произвольные объекты** `Blob`**. Свойство** `type` **объекта** `Blob` **в этом случае будет использоваться для установки заголовка «Content-Type», если он не будет определен явно. Если потребуется выгрузить двоичные данные, сгенерированные клиентским сценарием, можно воспользоваться приемами преобразования данных в объект** `Blob`**, демонстрируемыми в разделах 22.5 и 22.6.3, и передавать в виде тела запроса этот объект.**

18.1.3.5. Запросы с данными в формате multipart/form-data

Когда наряду с другими элементами HTML-формы включают элементы выгрузки файлов, броузер не может использовать обычный способ представления данных форм и должен отправлять формы, используя специальное значение «multipart/form-data» в заголовке «Content-Type». Этот формат связан с использованием длинных «граничных» строк, делящих тело запроса на несколько частей. Для текстовых данных можно вручную создать тело «multipart/form-data» запроса, но это довольно сложно.

Спецификация «XHR2» определяет новый прикладной интерфейс `FormData`**, упрощающий создание тела запроса, состоящего из нескольких частей. Сначала с помощью конструктора** `FormData()` **создается объект** `FormData`**, а затем вызовом метода** `append()` **этого объекта в него добавляются отдельные «части» (которые могут быть строками или объектами** `File` **и** `Blob`**). В заключение объект** `FormData` **передается методу** `send()`**. Метод** `send()` **определит соответствующую строку, обозначающую границу, и установит заголовок «Content-Type» запроса. Пример 18.10 демонстрирует использование объекта** `FormData`**, с которым мы еще встретимся в примере 18.11.**

Пример 18.10. Отправка запроса с данными в формате multipart/form-data

```
function postFormData(url, data, callback) {
    if (typeof FormData === "undefined")
```

```
        throw new Error("Объект FormData не реализован");

    var request = new XMLHttpRequest(); // Новый HTTP-запрос
    request.open("POST", url);          // Методом POST на указанный URL
    request.onreadystatechange = function() {     // Простой обработчик.
        if (request.readyState === 4 && callback) // При получении ответа
            callback(request);                    // вызвать указанную функц.
    };

    var formdata = new FormData();
    for(var name in data) {
        if (!data.hasOwnProperty(name)) continue; // Пропустить унасл. св-ва
        var value = data[name];
        if (typeof value === "function") continue; // Пропустить методы
        // Каждое свойство станет отдельной "частью" тела запроса.
        // Допускается использовать объекты File
        formdata.append(name, value);             // Добавить имя/значение,
    }                                             // как одну часть
    // Отправить пары имя/значение в теле запроса multipart/form-data. Каждая пара -
    // это одна часть тела запроса. Обратите внимание, что метод send автоматически
    // устанавливает заголовок Content-Type, когда ему передается объект FormData
    request.send(formdata);
}
```

18.1.4. События, возникающие в ходе выполнения HTTP-запроса

В примерах выше для определения момента завершения HTTP-запроса использовалось событие «readystatechange». Проект спецификации «XHR2» определяет более удобный набор событий, уже реализованный в Firefox, Chrome и Safari. В этой новой модели событий объект `XMLHttpRequest` генерирует различные типы событий на разных этапах выполнения запроса, благодаря чему отпадает необходимость проверять значение свойства `readyState`.

Далее описывается, как генерируются эти новые события в броузерах, поддерживающих их. Когда вызывается метод `send()`, один раз возбуждается событие «loadstart». В ходе загрузки ответа сервера объект `XMLHttpRequest` возбуждает серию событий «progress», обычно каждые 50 миллисекунд или около того, которые можно использовать для обратной связи с пользователем, чтобы информировать его о ходе выполнения запроса. Если запрос завершается очень быстро, событие «progress» может и не возбуждаться. По завершении запроса возбуждается событие «load».

Завершение запроса не всегда означает успешное его выполнение, поэтому обработчик события «load» должен проверять код состояния в объекте `XMLHttpRequest`, чтобы убедиться, что был принят HTTP-код «200 OK», а не «404 Not Found», например.

Существуют три разные ситуации, когда HTTP-запрос оканчивается неудачей, которым соответствуют три события. Если предельное время ожидания ответа истекло, генерируется событие «timeout». Если выполнение запроса было прервано, генерируется событие «abort». (О предельном времени ожидания и о методе `abort()` подробнее рассказывается в 18.1.5.) Наконец, выполнению запроса могут препятствовать другие ошибки в сети, такие как слишком большое количество переадресаций, и в этих случаях генерируется событие «error».

Для каждого запроса броузер может возбуждать только по одному событию «load», «abort», «timeout» и «error». Проект спецификации «XHR2» требует, чтобы броузеры возбуждали событие «loadend» после одного из этих событий. Однако на момент написания этих строк событие «loadend» не было реализовано ни в одном из броузеров.

Для регистрации обработчиков всех этих событий, возникающих в ходе выполнения запроса, можно использовать метод `addEventListener()` объекта `XMLHttpRequest`. Если каждое из этих событий обрабатывается единственным обработчиком, эти обработчики обычно проще установить, присвоив их соответствующим свойствам объекта, таким как `onprogress` и `onload`. Определяя наличие этих свойств, можно даже проверить поддержку соответствующих событий в броузере:

```
if ("onprogress" in (new XMLHttpRequest())) {
    // События, возникающие в ходе выполнения запроса, поддерживаются
}
```

Объект события, связанный с этими событиями, возникающими в ходе выполнения запроса, в дополнение к свойствам обычного объекта `Event`, таким как `type` и `timestamp`, добавляет три полезных свойства. Свойство `loaded` определяет количество байтов, переданных к моменту возбуждения события. Свойство `total` содержит общий объем (в байтах) загружаемых данных, определяемый из заголовка «Content-Length», или 0, если объем содержимого не известен. Наконец, свойство `lengthComputable` содержит значение `true`, если общий объем содержимого известен, и `false` – в противном случае. Очевидно, что свойства `total` и `loaded` особенно полезны в обработчиках событий, возникающих в ходе выполнения запроса:

```
request.onprogress = function(e) {
    if (e.lengthComputable)
        progress.innerHTML = Math.round(100*e.loaded/e.total) + "% Выполнено";
}
```

18.1.4.1. События, возникающие в ходе выгрузки

Кроме событий, которые удобно использовать для мониторинга загрузки HTTP-ответа, спецификация «XHR2» также определяет события для мониторинга выгрузки HTTP-запроса. В броузерах, реализующих эту возможность, объект `XMLHttpRequest` имеет свойство `upload`. Значением свойства `upload` является объект, определяющий метод `addEventListener()` и полный набор свойств-событий хода выполнения операции выгрузки, таких как `onprogress` и `onload`. (Однако этот объект не определяет свойство `onreadystatechange`: в процессе выгрузки генерируются только эти новые события.)

Обработчики событий хода выполнения операции выгрузки можно использовать точно так же, как используются обычные обработчики событий хода выполнения загрузки. Для объекта `x` типа `XMLHttpRequest` обработчик `x.onprogress` позволяет вести мониторинг хода выполнения загрузки ответа. А свойство `x.upload.onprogress` – мониторинг хода выполнения выгрузки запроса.

Пример 18.11 демонстрирует, как использовать событие «progress», генерируемое в ходе выгрузки запроса, для организации обратной связи с пользователем. Этот пример также демонстрирует, как получать объекты `File` с применением механизма буксировки (drag-and-drop) и как с помощью объекта `FormData` выгружать сразу несколько файлов в одном запросе `XMLHttpRequest`. На момент написания

этих строк спецификации, определяющие эти особенности, находились в состоянии проектирования и этот пример работал не во всех броузерах.

Пример 18.11. Мониторинг хода выполнения операции выгрузки

```
// Отыскивает все элементы с классом "fileDropTarget" и регистрирует
// обработчики событий механизма DnD, чтобы они могли откликаться
// на операции буксировки файлов. При сбросе файлов на эти элементы
// они выгружают их на URL-адрес, указанный в атрибуте data-uploadto.
whenReady(function() {
    var elts = document.getElementsByClassName("fileDropTarget");
    for(var i = 0; i < elts.length; i++) {
        var target = elts[i];
        var url = target.getAttribute("data-uploadto");
        if (!url) continue;
        createFileUploadDropTarget(target, url);
    }

    function createFileUploadDropTarget(target, url) {
        // Следит за ходом выполнения операции выгрузки и позволяет отвергнуть
        // выгрузку файла. Можно было бы обрабатывать сразу несколько параллельных
        // операций выгрузки, но это значительно усложнило бы
        // отображение хода их выполнения.
        var uploading = false;

        console.log(target, url);

        target.ondragenter = function(e) {
            console.log("dragenter");
            if (uploading) return; // Игнорировать попытку сброса, если
                                   // элемент уже занят выгрузкой файла
            var types = e.dataTransfer.types;
            if (types &&
                ((types.contains && types.contains("Files")) ||
                 (types.indexOf && types.indexOf("Files") !== -1))) {
                target.classList.add("wantdrop");
                return false;
            }
        };
        target.ondragover = function(e) { if (!uploading) return false; };
        target.ondragleave = function(e) {
            if (!uploading) target.classList.remove("wantdrop");
        };
        target.ondrop = function(e) {
            if (uploading) return false;
            var files = e.dataTransfer.files;
            if (files && files.length) {
                uploading = true;
                var message = "Выгружаются файлы:<ul>";
                for(var i = 0; i < files.length; i++)
                    message += "<li>" + files[i].name + "</li>";
                message += "</ul>";

                target.innerHTML = message;
                target.classList.remove("wantdrop");
                target.classList.add("uploading");
                var xhr = new XMLHttpRequest();
                xhr.open("POST", url);
                var body = new FormData();
                for(var i=0; i < files.length; i++) body.append(i, files[i]);
                xhr.upload.onprogress = function(e) {
                    if (e.lengthComputable) {
                        target.innerHTML = message +
                        Math.round(e.loaded/e.total*100) +
                        "% Завершено";
                    }
                };
                xhr.upload.onload = function(e) {
                    uploading = false;
                    target.classList.remove("uploading");
                    target.innerHTML = "Отбуксируйте сюда файл для выгрузки";
                };
                xhr.send(body);

                return false;
            }
            target.classList.remove("wantdrop");
        }
    }
});
```

18.1.5. Прерывание запросов и предельное время ожидания

Выполнение HTTP-запроса можно прерывать вызовом метода `abort()` объекта `XMLHttpRequest`. Метод `abort()` доступен во всех версиях объекта `XMLHttpRequest`, и согласно спецификации «XHR2» вызов метода `abort()` генерирует событие «abort». (На момент написания этих строк некоторые броузеры уже поддерживали событие «abort». Наличие этой поддержки можно определить по присутствию свойства `onabort` в объекте `XMLHttpRequest`.)

Основная причина для вызова метода `abort()` – появление необходимости отменить запрос, превышение предельного времени ожидания или если ответ становится ненужным. Допустим, что объект `XMLHttpRequest` используется для запроса подсказки в механизме автодополнения для текстового поля ввода. Если пользователь успеет ввести в поле новый символ еще до того, как подсказка будет получена с сервера, надобность в этой подсказке отпадает и запрос можно прервать.

Спецификация «XHR2» определяет свойство `timeout`, в котором указывается промежуток времени в миллисекундах, после которого запрос автоматически будет прерван, а также определяет событие «timeout», которое должно генерироваться (вместо события «abort») по истечении установленного промежутка времени. На момент написания этих строк броузеры еще не поддерживали автоматическое прерывание запроса по истечении предельного времени ожидания (и объекты `XMLHttpRequest` в них не имели свойств `timeout` и `ontimeout`). Однако имеется возможность реализовать собственную поддержку прерывания запросов по истечении заданного интервала времени с помощью функции `setTimeout()` (раздел 14.1) и метода `abort()`. Как это сделать, демонстрирует пример 18.12.

Пример 18.12. Реализация поддержки предельного времени ожидания

```
// Выполняет запрос HTTP GET на получение содержимого указанного URL.
// В случае благополучного получения ответа передает содержимое responseText функции
// обратного вызова. Если ответ не пришел в течение указанного времени, выполнение
// запроса прерывается. Броузеры могут возбуждать событие "readystatechange" после
// вызова abort(), а в случае получения части ответа свойство status может даже
// свидетельствовать об успехе, поэтому необходимо установить флаг, чтобы избежать
// вызова функции в случае получения части ответа при прерывании запроса по превышению
// времени ожидания. Эта проблема не наблюдается при использовании события load.
function timedGetText(url, timeout, callback) {
    var request = new XMLHttpRequest(); // Создать новый запрос.
    var timedout = false;               // Истекло ли время ожидания.
    // Запустить таймер, который прервет запрос по истечении
    // времени, заданного в миллесекундах.
    var timer = setTimeout(function() { // Запустить таймер. Если сработает,
                               timedout = true; // установить флаг
                               request.abort(); // и прервать запрос.
                           },
                           timeout);            // Интервал времени ожидания
    request.open("GET", url);                   // Указать URL запроса
    request.onreadystatechange = function() {   // Обработчик события.
        if (request.readyState !== 4) return;   // Игнорировать незаконч. запрос
        if (timedout) return;                   // Игнорировать прерв. запрос
        clearTimeout(timer);                    // Остановить таймер.
        if (request.status === 200)             // В случае успеха
            callback(request.responseText);     // передать ответ функции.
    };
    request.send(null);                         // Отправить запрос
}
```

18.1.6. Выполнение междоменных HTTP-запросов

Будучи субъектом политики общего происхождения (раздел 13.6.2), объект XMLHttpRequest может использоваться для отправки HTTP-запросов только серверу, откуда был получен использующий его документ. Это ограничение закрывает дыру безопасности, но также предотвращает возможность использования объекта для вполне законных междоменных запросов. Элементы <form> и <iframe> позволяют указывать URL-адреса с другими доменными именами, и в этих случаях броузеры будут отображать документы, полученные из других доменов. Но из-за ограничений политики общего происхождения броузер не позволит оригинальному сценарию исследовать содержимое стороннего документа. При использовании объекта XMLHttpRequest содержимое документа всегда доступно через свойство responseText, поэтому политика общего происхождения не позволяет объекту XMLHttpRequest выполнять междоменные запросы. (Обратите внимание, что элемент <script> в действительности никогда не был субъектом политики общего происхождения: он будет загружать и выполнять любые сценарии, независимо от их происхождения. Как будет показано в разделе 18.2, такая свобода выполнения междоменных запросов делает элемент <script> привлекательной альтернативой Ajax-транспорту на основе объекта XMLHttpRequest.)

Спецификация «XHR2» позволяет выполнять междоменные запросы к другим веб-сайтам, указанным в заголовке «CORS» (Cross-Origin Resource Sharing) HTTP-ответа. На момент написания этих строк текущие версии броузеров Firefox, Safari и Chrome поддерживали заголовок «CORS», а версия IE8 поддерживала собственный объект `XDomainRequest`, который здесь не описывается. Веб-программисту при этом не придется делать ничего особенного: если броузер поддерживает заголовок «CORS» для объекта `XMLHttpRequest` и если вы пытаетесь выполнить междоменный запрос к веб-сайту, доступ к которому разрешен в заголовке «CORS», политика общего происхождения просто не будет препятствовать выполнению междоменного запроса, и все будет работать само собой.

Хотя при наличии поддержки заголовка «CORS» не требуется предпринимать какие-то дополнительные шаги для выполнения междоменных запросов, тем не менее имеются некоторые детали, касающиеся безопасности, которые желательно понимать. Во-первых, при передаче имени пользователя и пароля методу `open()` объекта `XMLHttpRequest` они не будут передаваться в междоменных запросах (в противном случае это дало бы злоумышленникам возможность взламывать пароли). Кроме того, междоменные запросы обычно не допускают включение в них никакой другой информации, подтверждающей подлинность пользователя: cookies и HTTP-лексемы аутентификации обычно не передаются в составе запроса, а любые cookies, полученные в результате выполнения междоменного запроса, уничтожаются. Если для выполнения междоменного запроса требуется выполнить аутентификацию пользователя, перед вызовом метода `send()` следует установить свойство `withCredentials` объекта `XMLHttpRequest` в значение `true`. Проверка наличия свойства `withCredentials` позволит определить наличие поддержки заголовка «CORS» в броузере, хотя обычно этого не требуется.

В примере 18.13 приводится ненавязчивый JavaScript-код, использующий объект `XMLHttpRequest` для выполнения HEAD-запросов HTTP с целью получения информации о типах, размерах и датах последнего изменения ресурсов, на которые ссылаются элементы `<a>`. HEAD-запросы выполняются по требованию, а полученная информация отображается во всплывающих подсказках. В этом примере предполагается, что информация не будет доступна для междоменных ссылок, тем не менее в броузерах с поддержкой заголовка «CORS» все равно будут выполняться попытки получить ее.

Пример 18.13. Получение информации о ссылках с помощью HEAD-запросов при наличии поддержки заголовка «CORS»

```
/**
 * linkdetails.js
 *
 * Этот модуль в стиле ненавязчивого JavaScript отыскивает все элементы <a>
 * с атрибутом href и без атрибута title, и добавляет в них обработчики
 * события onmouseover. Обработчик события выполняет HEAD-запрос с помощью
 * объекта XMLHttpRequest, чтобы получить сведения о ресурсе, на который
 * указывает ссылка, и сохраняет эту информацию в атрибуте title ссылки,
 * благодаря чему эта информация будет отображаться во всплывающей подсказке.
 */
whenReady(function() {
    // Поддерживается ли возможность выполнения междоменных запросов?
    var supportsCORS = (new XMLHttpRequest()).withCredentials !== undefined;

    // Обойти в цикле все ссылки в документе
    var links = document.getElementsByTagName('a');
    for(var i = 0; i < links.length; i++) {
        var link = links[i];
        if (!link.href) continue; // Пропустить якоря, не являющиеся ссылками
        if (link.title) continue; // Пропустить ссылки с атрибутом title

        // Если это междоменная ссылка
        if (link.host!==location.host || link.protocol !== location.protocol)
        {
            link.title = "Ссылка на другой сайт"; // Предполагается, что нельзя
                                                  // получить дополнительную информацию
            if (!supportsCORS) continue;          // Пропустить, если заголовок
                                                  // CORS не поддерживается
            // Иначе есть надежда получить больше сведений о ссылке. Поэтому регистрируем
            // обработчик события, который предпримет попытку сделать это.
        }

        // Зарегистрировать обработчик события, который получит сведения
        // о ссылке при наведении на нее указателя мыши
        if (link.addEventListener)
            link.addEventListener("mouseover", mouseoverHandler, false);
        else
            link.attachEvent("onmouseover", mouseoverHandler);
    }

    function mouseoverHandler(e) {
        var link = e.target || e.srcElement;   // Элемент <a>
        var url = link.href;                   // URL-адрес ссылки

        var req = new XMLHttpRequest();        // Новый запрос
        req.open("HEAD", url);                 // Запросить только заголовки
        req.onreadystatechange = function() {  // Обработчик события
            if (req.readyState !== 4) return;  // Игнорировать незаверш. запросы
            if (req.status === 200) {          // В случае успеха
                var type = req.getResponseHeader("Content-Type");   //Получить
                var size = req.getResponseHeader("Content-Length"); //сведения
                var date = req.getResponseHeader("Last-Modified");  //о ссылке
                // Отобразить сведения во всплывающей подсказке.
                link.title = "Тип: " + type + " \n" +
                    "Размер: " + size + " \n" + "Дата: " + date;
            }
            else {
                // Если запрос не удался и подсказка для ссылки еще не содержит текст
                // "Ссылка на другой сайт", вывести сообщение об ошибке.
                if (!link.title)
                    link.title = "Невозможно получить сведения: \n" +
                        req.status + " " + req.statusText;
            }
        };
        req.send(null);

        // Удалить обработчик: попытка получить сведения выполняется только один раз.
        if (link.removeEventListener)
            link.removeEventListener("mouseover", mouseoverHandler, false);
        else
            link.detachEvent("onmouseover", mouseoverHandler);
    }
});
```

18.2. Выполнение HTTP-запросов с помощью <script>: JSONP

В начале этой главы упоминалось, что элемент `<script>` можно использовать в качестве Ajax-транспорта: достаточно установить атрибут `src` элемента `<script>` (и вставить его в документ, если он еще не внутри документа), и броузер сгенерирует HTTP-запрос, чтобы загрузить содержимое ресурса по указанному URL-адресу. Основная причина, почему элементы `<script>` являются удобным Ajax-транспортом, состоит в том, что они не являются субъектами политики общего происхождения и их можно использовать для запроса данных с других серверов. Вторая причина состоит в том, что элементы `<script>` автоматически декодируют (т. е. выполняют) тело ответа, содержащего данные в формате JSON.

Прием использования элемента `<script>` в качестве Ajax-транспорта известен под названием JSONP: это способ организации взаимодействий, когда в теле ответа на HTTP-запрос возвращаются данные в формате JSON. Символ «P» в аббревиатуре означает «padding», или «prefix» (дополнение или приставка), но об этом чуть ниже.[1]

Представьте, что вы пишете службу, которая обрабатывает GET-запросы и возвращает данные в формате JSON. Документы с общим происхождением могут пользоваться этой службой с помощью объекта `XMLHttpRequest` и метода `JSON.parse()`, как показано в примере 18.3. Если на сервере будет настроена отправка заголовка «CORS», сторонние документы при отображении в новых броузерах также смогут пользоваться службой с помощью объекта `XMLHttpRequest`. Однако при отображении в старых броузерах, не поддерживающих заголовок «CORS», сторонние документы смогут получить доступ к службе только с помощью элемента `<script>`. Ответ в формате JSON является (по определению) допустимым программным кодом на языке JavaScript, и броузер выполнит его при получении. При выполнении данных в формате JSON происходит их декодирование, но полученный результат является обычными данными, которые ничего не *делают*.

Именно здесь на сцену выходит символ «P» из аббревиатуры JSONP. Когда обращение к службе реализовано с помощью элемента `<script>`, служба должна «дополнить» ответ, окружив его круглыми скобками и добавив в начало имя JavaScript-функции. То есть вместо того чтобы отправлять данные, такие как:

```
[1, 2, {"buckle": "my shoe"}]
```

она должна отправлять дополненные данные, как показано ниже:

```
handleResponse(
[1, 2, {"buckle": "my shoe"}]
)
```

Являясь телом элемента `<script>`, этот дополненный ответ уже будет выполнять некоторые действия: он произведет преобразование данных из формата JSON (которые в конечном итоге являются одним длинным выражением на языке JavaScript) и передаст их функции `handleResponse()`, которая, как предполагается, определена в теле документа и выполняет некоторые операции с данными.

---

[1] Термин «JSONP» предложил Боб Ипполито (Bob Ippolito) в 2005 году (*http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/*).

Сценарии и безопасность

Используя элемент `<script>` в качестве Ajax-транспорта, вы разрешаете своей веб-странице выполнять любой программный код на языке JavaScript, который отправит удаленный сервер. Это означает, что прием, описываемый здесь, не должен использоваться при работе с серверами, не вызывающими доверия. Впрочем, даже при работе с доверенным сервером следует помнить, что этот сервер может быть взломан злоумышленником, и злоумышленник сможет заставить вашу веб-страницу выполнить любой программный код и отобразить любую информацию, какую он пожелает, и эта информация будет выглядеть, как если бы она поступила с вашего сайта.

При этом следует отметить, что для веб-сайтов стало обычным делом использовать доверенные сценарии сторонних разработчиков, особенно сценарии, используемые для внедрения в страницу рекламы или «виджетов». Использование элемента `<script>` в качестве Ajax-транспорта для взаимодействия с доверенными веб-службами ничуть не опаснее.

Чтобы этот прием действовал, необходимо иметь некоторый способ сообщить службе, что обращение к ней выполняется с помощью элемента `<script>` и она должна возвращать ответ не в формате JSON, а в формате JSONP. Это можно сделать, указав параметр запроса в адресе URL, например, добавив в конец `?jsonp` (или `&jsonp`).

На практике веб-службы, поддерживающие JSONP, не диктуют имя функции, такое как «handleResponse», которую должны реализовать все клиенты. Вместо этого они используют значение параметра запроса, позволяя клиентам самим указывать имя функции, и применяют его для дополнения ответа. В примере 18.14 для определения имени функции обратного вызова используется параметр «jsonp». Многие службы, поддерживающие JSONP, распознают параметр с эти именем. Также достаточно часто используется параметр с именем «callback», и вы можете изменить программный код примера, чтобы он смог работать со службами, предъявляющими определенные требования.

Пример 18.14 определяет функцию `getJSONP()`, которая выполняет запрос на получение данных в формате JSONP. В этом примере есть несколько тонкостей, о которых следует сказать особо. Во-первых, обратите внимание, что пример создает новый элемент `<script>`, устанавливает его URL-адрес и вставляет элемент в документ. Операция вставки вызывает выполнение HTTP-запроса. Во-вторых, для каждого запроса в этом примере создается новая внутренняя функция обратного вызова, которая сохраняется в свойстве самой функции `getJSONP()`. Наконец, эта функция обратного вызова выполняет необходимые заключительные операции: она удаляет элемент `<script>` и саму себя.

Пример 18.14. Выполнение запросов на получение данных в формате JSONP с помощью элемента <script>

```
// Выполняет запрос по указанному URL-адресу на получение данных в формате JSONP и передает
// полученные данные указанной функции обратного вызова. Добавляет в URL параметр запроса
// с именем "jsonp", чтобы указать имя функции обратного вызова.
function getJSONP(url, callback) {
    // Создать для данного запроса функцию с уникальным именем
    var cbnum = "cb" + getJSONP.counter++; // Каждый раз увеличивать счетчик
    var cbname = "getJSONP." + cbnum;      // Как свойство этой функции

    // Добавить имя функции в строку запроса url, используя формат представления данных
    // HTML-форм. Здесь используется параметр с именем "jsonp". Некоторые веб-службы
    // с поддержкой JSONP могут использовать параметр с другим именем, таким как "callback".
    if (url.indexOf("?") === -1)   // URL еще не имеет строки запроса
        url += "?jsonp=" + cbname; // добавить параметр как строку запроса
    else                           // В противном случае
        url += "&jsonp=" + cbname; // добавить как новый параметр.

    // Создать элемент script, который отправит запрос
    var script = document.createElement("script");

    // Определить функцию, которая будет вызвана сценарием
    getJSONP[cbnum] = function(response) {
        try {
            callback(response);                   // Обработать данные
        }
        finally { // Даже если функция или ответ возбудит исключение
            delete getJSONP[cbnum];               // Удалить эту функцию
            script.parentNode.removeChild(script); // Удалить элемент script
        }
    };

    // Инициировать HTTP-запрос
    script.src = url;                   // Указать url-адрес элемента
    document.body.appendChild(script); // Добавить в документ
}

getJSONP.counter = 0; // Счетчик, используемый для создания уникальных имен
```

18.3. Архитектура Comet на основе стандарта «Server-Sent Events»

Проект стандарта «Server-Sent Events» определяет объект `EventSource`, который делает практически тривиальным создание приложений с архитектурой Comet. При его использовании достаточно передать URL-адрес конструктору `EventSource()` и затем обрабатывать события «message» в полученном объекте:

```
var ticker = new EventSource("stockprices.php");
ticker.onmessage = function(e) {
    var type = e.type;
    var data = e.data;
    // Обработать строки type и data.
}
```

Объект события «message» имеет свойство `data`, хранящее строку, отправленную сервером с этим событием в качестве полезной нагрузки. Кроме того, объект события имеет свойство `type`, как и все другие объекты событий. По умолчанию это свойство имеет значение «message», но источник события может указать в этом свойстве другую строку. Все события от данного сервера, источника событий, обрабатываются единственным обработчиком `onmessage`, который при необходимости может передавать их другим обработчикам, опираясь на свойство `type` объекта события.

Протокол обмена, определяемый стандартом «Server-Sent Event», достаточно прост. Клиент устанавливает соединение с сервером (когда создает объект EventSource**), а сервер сохраняет это соединение открытым. Когда происходит событие, сервер передает через соединение текстовую строку. Передача события через сеть выглядит примерно следующим образом:**

`event: bid`	***установка свойства*** `type` ***объекта события***
`data: GOOG`	***установка свойства*** `data`
`data: 999`	***добавляется перевод строки и дополнительные данные***
	пустая строка генерирует событие `message`

Протокол имеет также некоторые дополнительные особенности, позволяющие присваивать событиям идентификаторы и дающие клиенту возможность после восстановления соединения с сервером передавать этот идентификатор, чтобы сервер мог повторно послать все события, пропущенные клиентом. Однако эти особенности не имеют большого значения в данном обсуждении.

Одно из очевидных применений архитектуры Comet – реализация чатов: клиент может посылать в чат новые сообщения с помощью объекта XMLHttpRequest **и подписываться на поток сообщений, поступающих от собеседников, с помощью объекта** EventSource**. Пример 18.15 демонстрирует, насколько просто реализовать клиента на основе объекта** EventSource**.**

Пример 18.15. Простой клиент чата на основе объекта EventSource

```
<script>
window.onload = function() {
    // Позаботиться о некоторых особенностях пользовательского интерфейса
    var nick = prompt("Введите ваше имя");        // Получить имя пользователя
    var input = document.getElementById("input"); // Отыскать поле ввода
    input.focus();                                // Передать фокус ввода

    // Подписаться на получение новых сообщений с помощью объекта EventSource
    var chat = new EventSource("/chat");
    chat.onmessage = function(event) {           // Получив новое сообщение,
        var msg = event.data;                    // Извлечь текст
        var node = document.createTextNode(msg); // Преобр. в текстовый узел
        var div = document.createElement("div"); // Создать <div>
        div.appendChild(node);                   // Добавить текст. узел в div
        document.body.insertBefore(div, input);  // И добавить div перед input
        input.scrollIntoView();                  // Прокрутить до появления
    }                                            // input в видимой области

    // Передавать сообщения пользователя на сервер с помощью XMLHttpRequest
    input.onchange = function() {                // При нажатии клавиши Enter
        var msg = nick + ": " + input.value;     // Имя пользователя + введ. текст
        var xhr = new XMLHttpRequest();          // Создать новый XHR
        xhr.open("POST", "/chat");               // POST-запрос к /chat.
        xhr.setRequestHeader("Content-Type",     // Тип - простой текст UTF-8
                             "text/plain;charset=UTF-8");
        xhr.send(msg);                           // Отправить сообщение
        input.value = "";                        // Приготовиться к вводу
    }                                            // следующего сообщения
};
</script>
<!-- Пользовательский интерфейс чата состоит из единственного поля ввода -->
```

```
<!-- Новые сообщения, отправленные в чат, вставляются перед полем ввода -->
<input id="input" style="width:100%"/>
```

На момент написания этих строк объект `EventSource` **поддерживался в Chrome и Safari и должен был быть реализован компанией Mozilla в первом же выпуске Firefox, вышедшем после версии 4.0. В броузерах (таких как Firefox), где реализация объекта** `XMLHttpRequest` **возбуждает событие «readystatechange» в ходе загрузки ответа (для значения 3 в свойстве** `readyState`**) многократно, поведение объекта** `EventSource` **относительно легко имитировать с помощью объекта** `XMLHttpRequest`**, как демонстрируется в примере 18.16. С модулем имитации пример 18.15 будет работать в Chrome, Safari и Firefox. (Пример 18.16 не будет работать в броузерах IE и Opera, поскольку реализации объекта** `XMLHttpRequest` **в этих броузерах не генерируют события в ходе загрузки.)**

Пример 18.16. Имитация объекта EventSource с помощью XMLHttpRequest

```
// Имитация прикладного интерфейса EventSource в броузерах, не поддерживающих его.
// Требует, чтобы XMLHttpRequest генерировал многократные события readystatechange
// в ходе загрузки данных из долгоживущих HTTP-соединений. Учтите, что это не полноценная
// реализация API: она не поддерживает свойство readyState, метод close(), а также
// события open и error. Кроме того, регистрация обработчика события message выполняется
// только через свойство onmessage -- эта версия не определяет метод addEventListener
if (window.EventSource === undefined) {       // Если EventSource не поддерживается,
    window.EventSource = function(url) {      // использовать эту его имитацию.
        var xhr;                              // HTTP-соединение...
        var evtsrc = this;         // Используется в обработчиках.
        var charsReceived = 0;     // Так определяется появление нового текста
        var type = null;           // Для проверки свойства type ответа.
        var data = "";             // Хранит данные сообщения
        var eventName = "message"; // Значение поля type объектов событий
        var lastEventId = "";      // Для синхронизации с сервером
        var retrydelay = 1000;     // Задержка между попытками соединения
        var aborted = false;       // Установите в true, чтобы разорвать соединение

        // Создать обьект XHR
        xhr = new XMLHttpRequest();

        // Определить обработчик события для него
        xhr.onreadystatechange = function() {
            switch(xhr.readyState) {
            case 3: processData(); break;     // При получении фрагмента данных
            case 4: reconnect(); break;       // По завершении запроса
            }
        };

        // И установить долгоживущее соединение
        connect();

        // Если соединение было закрыто обычным образом, ждать секунду
        // и попробовать восстановить соединение
        function reconnect() {
            if (aborted) return;              // Не восстанавливать после
                                              // принудительного прерывания
            if (xhr.status >= 300) return;    // Не восстанавливать после ошибки
            setTimeout(connect, retrydelay);  // Ждать и повторить попытку
        };
```

```
// Устанавливает соединение
function connect() {
    charsReceived = 0;
    type = null;
    xhr.open("GET", url);
    xhr.setRequestHeader("Cache-Control", "no-cache");
    if (lastEventId)
        xhr.setRequestHeader("Last-Event-ID", lastEventId);
    xhr.send();
}

// При получении данных обрабатывает их и вызывает обработчик onmessage.
// Эта функция реализует работу с протоколом Server-Sent Events
function processData() {
    if (!type) { // Проверить тип ответа, если это еще не сделано
        type = xhr.getResponseHeader('Content-Type');
        if (type !== "text/event-stream") {
            aborted = true;
            xhr.abort();
            return;
        }
    }
    // Запомнить полученный объем данных и извлечь только ту часть ответа,
    // которая еще не была обработана.
    var chunk = xhr.responseText.substring(charsReceived);
    charsReceived = xhr.responseText.length;

    // Разбить текст на строки и обойти их в цикле.
    var lines = chunk.replace(/(\r\n|\r|\n)$/,"").split(/\r\n|\r|\n/);
    for(var i = 0; i < lines.length; i++) {
        var line = lines[i], pos = line.indexOf(":"), name, value="";
        if (pos == 0) continue;     // Игнорировать комментарии
        if (pos > 0) {              // поле name:value
            name = line.substring(0,pos);
            value = line.substring(pos+1);
            if (value.charAt(0) == " ") value = value.substring(1);
        }
        else name = line;           // только поле name

        switch(name) {
        case "event": eventName = value; break;
        case "data": data += value + "\n"; break;
        case "id": lastEventId = value; break;
        case "retry": retrydelay = parseInt(value) || 1000; break;
        default: break;             // Игнорировать любые другие строки
        }

        if (line === "") { // Пустая строка означает отправку события
            if (evtsrc.onmessage && data !== "") {
                // Отсечь завершающий символ перевода строки
                if (data.charAt(data.length-1) == "\n")
                    data = data.substring(0, data.length-1);
                evtsrc.onmessage({  // Имитация объекта Event
                    type: eventName, // тип события
                    data: data,      // данные
                    origin: url      // происхождение данных
```

```
                        });
                    }
                    data = "";
                    continue;
                }
            }
        }
    };
}
```

Завершим описание архитектуры Comet примером серверного сценария. В примере 18.17 приводится реализация HTTP-сервера на серверном JavaScript, который выполняется под управлением интерпретатора Node (раздел 12.2). Когда клиент обращается к корневому URL «/», сервер отправляет ему реализацию клиента чата, представленную в примере 18.15, и реализацию имитации, представленную в примере 18.16. Когда клиент выполняет GET-запрос по URL-адресу «/chat», сервер сохраняет поток ответа в массиве и поддерживает соединение открытым. А когда клиент выполняет POST-запрос к адресу «/chat», сервер интерпретирует тело запроса как текст сообщения и добавляет префикс «data:», как того требует протокол Server-Sent Events, во все открытые потоки сообщений. Если вы установите интерпретатор Node, вы сможете запустить этот пример сервера локально. Он прослушивает порт 8000, поэтому после запуска сервера в броузере необходимо будет указать адрес *http://localhost:8000*, чтобы соединиться с сервером и начать общение с самим собой.

Пример 18.17. Сервер чата, поддерживающий протокол Server-Sent Events

```
// Этот программный код на серверном JavaScript предназначен для выполнения
// под управлением NodeJS. Он реализует очень простую, полностью анонимную комнату чата.
// Для отправки новых сообщений в чат следует использовать POST-запросы к URL /chat,
// а для получения текста/потока-событий сообщений следует использовать GET-запросы
// к тому же URL. При выполнении GET-запроса к / возвращается простой HTML-файл,
// содержащий пользовательский интерфейс чата для клиента.
var http = require('http'); // Реализация API HTTP-сервера в NodeJS

// HTML-файл для клиента чата. Используется ниже.
var clientui = require('fs').readFileSync("chatclient.html");
var emulation = require('fs').readFileSync("EventSourceEmulation.js");

// Массив объектов ServerResponse, который будет использоваться для отправки событий
var clients = [];

// Отправлять комментарий клиентам каждые 20 секунд, чтобы предотвратить
// закрытие соединения с последующими попытками восстановить его
setInterval(function() {
    clients.forEach(function(client) {
        client.write(":ping\n");
    });
}, 20000);

// Создать новый сервер
var server = new http.Server();

// Когда сервер получит новый запрос, он вызовет эту функцию
server.on("request", function (request, response) {
    // Проанализировать запрошенный URL
    var url = require('url').parse(request.url);
    // Если запрошен URL "/", отправить пользовательский интерфейс чата.
    if (url.pathname === "/") { // Запрошен пользовательский интерфейс чата
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write("<script>" + emulation + "</script>");
        response.write(clientui);
        response.end();
        return;
    }
    // Если запрошен любой другой URL, кроме "/chat", отправить код 404
    else if (url.pathname !== "/chat") {
        response.writeHead(404);
        response.end();
        return;
    }

    // Если был выполнен POST-запрос - клиент отправил новое сообщение
    if (request.method === "POST") {
        request.setEncoding("utf8");
        var body = "";
        // При получении фрагмента данных добавить его в переменную body
        request.on("data", function(chunk) { body += chunk; });

        // По завершении запроса отправить пустой ответ
        // и широковещательное сообщение всем клиентам.
        request.on("end", function() {
            response.writeHead(200); // Ответ на запрос
            response.end();

            // Преобразовать сообщение в формат текст/поток-событий.
            // Убедиться, что все строки начинаются с префикса "data:"
            // и само сообщение завершается двумя символами перевода строки.
            message = 'data: ' + body.replace('\n', '\ndata: ') + "\r\n\r\n";
            // Отправить сообщение всем клиентам
            clients.forEach(function(client) { client.write(message); });
        });
    }
    // Если иначе, клиент запросил поток сообщений
    else {
        // Установить тип содержимого и отправить начальное событие message
        response.writeHead(200, {'Content-Type': "text/event-stream" });
        response.write("data: Connected\n\n");

        // Если клиент закрыл соединение, удалить соответствующий
        // объект response из массива активных клиентов
        request.connection.on("end", function() {
            clients.splice(clients.indexOf(response), 1);
            response.end();
        });

        // Запомнить объект response, чтобы в дальнейшем посылать сообщения с его помощью
        clients.push(response);
    }
});

// Запустить сервер на порту 8000. Чтобы подключиться к нему, используйте
// адрес http://localhost:8000/
server.listen(8000);
```

19

Библиотека jQuery

В языке JavaScript чрезвычайно простой базовый и весьма сложный клиентский API, который к тому же отягощен многочисленными несовместимостями между броузерами. С выходом IE9 исчезли самые неприятные несовместимости, но многие программисты считают, что веб-приложения удобнее писать с использованием фреймворков или вспомогательных библиотек на языке JavaScript, упрощающих решение типичных задач и скрывающих различия между броузерами. На момент написания этих строк одной из наиболее популярных и широко используемых была библиотека jQuery.[1]

Библиотека jQuery получила весьма широкое распространение, и поэтому веб-разработчики должны быть знакомы с ней: даже если вы не собираетесь ее использовать, вы наверняка встретитесь с ней в сценариях, написанных другими. К счастью, библиотека jQuery весьма стабильна и достаточно мала, чтобы ее можно было описать в этой книге. В этой главе дается обстоятельное введение, а в четвертой части вы найдете краткий справочник по библиотеке jQuery. В справочнике отсутствуют отдельные статьи для методов из библиотеки jQuery, однако в статье `jQuery` даются краткие описания всех методов.

Библиотека jQuery упрощает поиск элементов документа и облегчает манипулирование ими: добавление содержимого, изменение HTML-атрибутов и CSS-свойств, определение обработчиков событий и воспроизведение анимационных эффектов. Она также имеет вспомогательные функции поддержки архитектуры Ajax, позволяющие выполнять динамические HTTP-запросы, и функции общего назначения для работы с объектами и массивами.

Как следует из ее имени, основу библиотеки jQuery составляет реализация механизма *запросов*. Типичный запрос использует CSS-селектор, идентифицирующий множество элементов документа, и возвращает объект, представляющий эти элементы. Данный возвращаемый объект имеет множество удобных методов для выполнения операций над всей группой выбранных элементов. Всякий раз, когда это возможно, эти методы возвращают объект, относительно которого они вызывались, что позволяет использовать прием составления цепочек из вызовов методов. Ниже перечислены особенности, которые обеспечивают широту возможностей и удобство использования библиотеки jQuery:

---

[1] В число широко используемых входят также не описываемые в этой книге библиотеки Prototype, YUI и dojo. Еще большее количество библиотек можно отыскать в Интернете, выполнив поиск по фразе «JavaScript libraries».

- Выразительный синтаксис (CSS-селекторов) для ссылок на элементы в документе
- Эффективная реализация механизма запросов, выполняющего поиск множества элементов документа, соответствующих указанному CSS-селектору
- Множество удобных методов для манипулирования множествами выбранных элементов
- Мощные приемы функционального программирования для выполнения операций сразу над всей группой элементов
- Выразительная идиома представления последовательностей операций (составление цепочек из вызовов методов)

Эта глава начинается с введения в библиотеку jQuery, где будет показано, как выполнять простейшие запросы и как обрабатывать результаты. В последующих разделах описывается:

- Как изменять HTML-атрибуты, стили и классы CSS, значения и содержимое элементов HTML-форм, управлять геометрией и данными
- Как изменять структуру документа, вставляя, замещая, обертывая и удаляя элементы
- Как использовать модель событий библиотеки jQuery, совместимую со всеми броузерами
- Как с помощью jQuery воспроизводить анимационные визуальные эффекты
- Как выполнять HTTP-запросы с помощью механизма поддержки архитектуры Ajax, реализованном в библиотеке jQuery
- Вспомогательные функции jQuery
- Полный синтаксис селекторов jQuery, и как использовать расширенные методы поиска из библиотеки jQuery
- Как расширять библиотеку jQuery, создавая и используя расширения
- Библиотека jQuery UI

19.1. Основы jQuery

Библиотека jQuery определяет единственную глобальную функцию с именем `jQuery()`. Эта функция используется настолько часто, что библиотека определяет также глобальное имя $, как сокращенный псевдоним этой функции. Эти два имени – все, что библиотека jQuery добавляет в глобальное пространство имен.[1]

---

[1] Если вы определяете собственное имя $ в своих сценариях или используете другую библиотеку, такую как Prototype, тоже использующую имя $, вы можете вызвать метод `jQuery.noConflict()`, чтобы восстановить оригинальное значение имени $.

Эта глобальная функция с двумя именами является центральной функцией механизма запросов в библиотеке jQuery. Например, ниже показано, как можно запросить множество всех элементов `<div>` в документе:

```
var divs = $("div");
```

Эта функция возвращает множество из нуля или более элементов DOM, которое называется объектом `jQuery`. Обратите внимание, что `jQuery()` является фабричной функцией, а не конструктором: она возвращает вновь созданный объект, но она используется без ключевого слова `new`. Объект `jQuery` определяет множество методов для выполнения операций над множеством элементов, которое он представляет, и большая часть главы будет посвящена описанию этих методов. Ниже приводится пример инструкции, которая отыскивает, выделяет цветом и быстро отображает все скрытые элементы `<p>`, имеющие класс «details»:

```
$("p.details").css("background-color", "yellow").show("fast");
```

Метод `css()` оперирует объектом `jQuery`, возвращаемым функцией `$()`, и возвращает этот же объект, благодаря чему метод `show()` может быть вызван в этой же компактной «цепочке вызовов методов». Этот прием составления цепочек является весьма характерным при использовании библиотеки jQuery. В качестве другого примера ниже демонстрируется инструкция, выполняющая поиск всех элементов, имеющих CSS-класс «clicktohide», и регистрирующая обработчик события в каждом из них. Этот обработчик вызывается, когда пользователь щелкает на элементе и заставляет элемент медленно «выезжать» за границы окна вверх и скрываться:

```
$(".clicktohide").click(function() { $(this).slideUp("slow"); });
```

19.1.1. Функция jQuery()

Функция `jQuery()` (она же `$()`) является наиболее важной в библиотеке jQuery. Однако она существенно перегружена и может быть вызвана четырьмя разными способами.

Первый и наиболее типичный способ вызова `$()` предусматривает передачу ей CSS-селектора (строки). При вызове таким способом она возвращает множество элементов из текущего документа, соответствующих селектору. Библиотека в значительной степени поддерживает синтаксис селекторов CSS3 плюс некоторые собственные расширения. Более подробное описание синтаксиса селекторов jQuery приводится в разделе 19.8.1. Если во втором аргументе передать функции `$()` элемент или объект `jQuery`, она вернет только элементы-потомки указанного элемента или элементов, соответствующие селектору. Этот необязательный второй аргумент определяет начальную точку (или точки) выполнения запроса и часто называется *контекстом*.

При втором способе вызова функции `$()` передается объект `Element`, `Document` или `Window`. Подобный вызов просто обертывает элемент, документ или окно объектом `jQuery` и возвращает его. Это дает возможность манипулировать элементом с помощью методов объекта `jQuery` вместо низкоуровневых методов модели DOM. Например, в программах, использующих библиотеку jQuery, часто можно встретить вызов `$(document)` или `$(this)`. Объекты `jQuery` могут представлять множество элементов документа, а кроме того, функции `$()` можно передавать массив элементов. В этом случае возвращаемый объект `jQuery` будет представлять множество элементов, имевшихся в массиве.

Получение библиотеки jQuery

Библиотека jQuery является свободно распространяемым программным обеспечением. Ее можно загрузить с сайта *http://jquery.com*. Получив библиотеку, вы сможете подключать ее к своим веб-страницам с помощью элемента `<script>`, как показано ниже:

```
<script src="jquery-1.4.2.min.js"></script>
```

Слово «min» в имени файла выше указывает, что используется минимизированная версия библиотеки, из которой были удалены комментарии и лишние пробелы, а внутренние идентификаторы были заменены более короткими именами.

Другой способ задействовать библиотеку jQuery в своих веб-приложениях заключается в использовании сети распространения библиотеки, для чего достаточно указать любой из следующих URL-адресов:

```
http://code.jquery.com/jquery-1.4.2.min.js
http://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.min.js
http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js
```

В этой главе описывается библиотека jQuery версии 1.4. Если вы пользуетесь другой версией, замените номер версии «1.4.2» в URL-адресах выше на требуемый вам.[1] Если вы пользуетесь сетью распространения Google CDN, вы можете указать номер версии «1.4», чтобы получить самую свежую версию в ветке 1.4.x, или просто «1», чтобы получить самую свежую версию ниже 2.0. Основное преимущество использования подобных широко известных адресов состоит в том, что благодаря огромной популярности jQuery посетители вашего веб-сайта наверняка будут иметь копию библиотеки в кэше своих броузеров, и веб-приложению не придется тратить дополнительное время на ее загрузку.

При третьем способе вызова функции `$()` передается строка с разметкой HTML. В этом случае библиотека jQuery создаст HTML-элемент или элементы, определяемые этой разметкой, и вернет представляющий их объект `jQuery`. Библиотека jQuery не вставляет вновь созданные элементы в документ, но методы объекта `jQuery`, описываемые в разделе 19.3, позволяют легко вставить их в любое место. Обратите внимание, что таким образом функции `$()` нельзя передать простой текст, так как в этом случае jQuery решит, что вы передали CSS-селектор. При таком способе вызова строка, передаваемая функции `$()`, должна включать хотя бы один HTML-тег в угловых скобках.

При вызове третьим способом функция `$()` может принимать необязательный второй аргумент. В нем можно передать объект `Document`, чтобы указать документ, с которым должны быть связаны элементы. (Если, к примеру, предполагается, что создаваемые элементы будут вставлены в элемент `<iframe>`, необходимо явно указать объект документа этого фрейма.) Или передать объект во втором аргументе. В этом случае предполагается, что свойства объекта определяют имена и значения HTML-атрибутов, которые должны быть установлены во вновь созданном элементе. Но если объект будет содержать свойства с такими именами, как «css», «html», «text», «width», «height», «offset», «val» или «data», или свойства с именами, совпадающими с именами методов регистрации обработчиков событий в библиотеке jQuery, она будет вызывать методы с этими именами для вновь созданного элемента и передавать им значения соответствующих свойств. (Методы, такие как `css()`, `html()` и `text()` рассматриваются в разделе 19.2, а методы регистрации обработчиков событий – в разделе 19.4). Например:

---

[1] На момент написания этой главы текущей версией библиотеки jQuery была 1.4.2. Когда книга готовилась к печати, была выпущена версия jQuery 1.5. Изменения в jQuery 1.5 в основном касаются механизма поддержки архитектуры Ajax и будут упоминаться в разделе 19.6.

```
var img = $("<img/>",              // Создать новый элемент <img>
            { src:url,             // с этим HTML-атрибутом,
              css: {borderWidth:5}, // этим CSS-стилем
              click: handleClick   // и этим обработчиком события.
            });
```

Наконец, при четвертом способе вызова функции `$()` передается функция. В этом случае указанная вами функция будет вызвана, когда документ будет полностью загружен и дерево DOM документа будет готово к выполнению операций. Это версия функции `onLoad()`, представленной в примере 13.5, реализованная в библиотеке jQuery. Очень часто можно встретить jQuery-программы, реализованные в виде анонимных функций, объявляемых в вызове функции `jQuery()`:

```
jQuery(function() { // Будет вызвана по окончании загрузки документа
    // Здесь находится весь программный код, использующий jQuery
});
```

Иногда можно увидеть вызов `$(f)`, оформленный в старом и более явном стиле:

```
$(document).ready(f).
```

Функция, передаваемая `jQuery()`, будет вызвана со ссылкой `this`, указывающей на объект `document`, и функцией `jQuery` в качестве единственного аргумента. Это означает, что вы можете удалить глобальное определение функции `$` и по-прежнему использовать этот удобный псевдоним локально, как показано ниже:

```
jQuery.noConflict(); // Восстановить оригинальное значение $
jQuery(function($) { // Использовать $ как локальный псевдоним функции jQuery
    // Здесь находится весь программный код, использующий jQuery
});
```

Функции, зарегистрированные с помощью `$()`, будут вызваны библиотекой jQuery, когда будет возбуждено событие «DOMContentLoaded» (раздел 13.3.4) или, если это событие не поддерживается, когда будет возбуждено событие «load». То есть когда документ будет загружен и полностью разобран, но внешние ресурсы, такие как изображения, еще могут быть не загружены. Если функцию передать в вызов `$()` после того, как дерево DOM будет готово, она будет вызвана немедленно, перед тем как `$()` вернет управление.

Кроме того, библиотека jQuery использует функцию `jQuery()` как собственное пространство имен и определяет в нем множество вспомогательных функций и свойств. Одной из таких вспомогательных функций является функция `jQuery.noConflict()`, упоминавшаяся выше. В числе других функций общего назначения можно назвать функцию `jQuery.each()`, предназначенную для выполнения итераций, `jQuery.parseJSON()` – для синтаксического анализа данных в формате JSON. Перечень вспомогательных функций общего назначения приводится в разделе 19.7, а другие вспомогательные функции из библиотеки jQuery описываются на протяжении всей главы.

Терминология jQuery

Давайте остановимся, чтобы определить некоторые важные термины, которые будут встречаться на протяжении этой главы:

«функция jQuery»

Функция jQuery – это значение `jQuery` или $. Эта функция создает объекты `jQuery`, регистрирует обработчики, которые вызываются, когда дерево DOM будет готово к выполнению операций, а также служит пространством имен библиотеки jQuery. Я обычно использую имя `$()`. Поскольку она служит пространством имен, функция jQuery может также называться «глобальным объектом jQuery», но очень важно не путать ее с «объектом `jQuery`».

«объект jQuery»

Объект `jQuery` – это объект, возвращаемый функцией jQuery. Объект `jQuery` представляет множество элементов документа и может также называться «результатом функции jQuery», «множеством jQuery» или «обернутым набором».

«выбранные элементы»

Когда функции jQuery передается CSS-селектор, она возвращает объект `jQuery`, представляющий множество элементов документа, соответствующих этому селектору. При описании методов объекта `jQuery` я часто буду употреблять фразу «выбранные элементы», ссылаясь на элементы множества. Например, при описании метода `attr()` я мог бы сказать: «метод `attr()` устанавливает HTML-атрибуты выбранных элементов». Вместо более точной, но трудно читаемой фразы: «метод `attr()` устанавливает HTML-атрибуты элементов в объекте `jQuery`, относительно которого он был вызван». Обратите внимание, что слово «выбранных» относится к CSS-селектору и не имеет никакого отношения к элементам, выбираемым пользователем.

«функция библиотеки jQuery»

Функция библиотеки jQuery – это функция, такая как `jQuery.noConflict()`, которая определена в пространстве имен функции jQuery. Функции библиотеки jQuery могут также упоминаться как «статические методы».

«метод jQuery»

Метод jQuery – это метод объекта `jQuery`, возвращаемого функцией jQuery. Наиболее важной частью библиотеки jQuery являются мощные методы, которые она определяет.

Иногда сложно уловить различия между методами объекта `jQuery` и функциями библиотеки jQuery, потому что многие методы и функции имеют одинаковые имена. Обратите внимание, что следующие две строки программного кода выполняют разные операции:

```
// Вызвать функцию each() библиотеки jQuery, чтобы вызвать функцию f
// для каждого элемента массива a
$.each(a,f);

// Вызвать функцию jQuery(), чтобы получить объект jQuery, представляющий
// все элементы <a> в документе. Затем вызвать метод each() этого объекта jQuery,
// чтобы вызвать функцию f для каждого выбранного элемента.
$("a").each(f);
```

В официальной документации по библиотеке jQuery, которую можно найти на сайте *http://jquery.com*, такие имена, как `$.each`, используются для ссылки на функции библиотеки jQuery, а такие имена, как `.each` (с точкой, но без знака доллара), – для ссылки на методы объекта `jQuery`. Вместо них в этой книге я буду использовать термины «функция» и «метод». Что именно подразумевается, обычно будет достаточно очевидно из контекста обсуждения.

19.1.2. Запросы и результаты запросов

Когда функции `$()` передается CSS-селектор, она возвращает объект `jQuery`, представляющий множество («выбранных») элементов, соответствующих селектору. С CSS-селекторами мы познакомились в разделе 15.2.5, куда вы можете вернуться, чтобы освежить память, – все примеры селекторов, представленные там, могут передаваться функции `$()`. Конкретный синтаксис селекторов, поддерживаемый библиотекой jQuery, подробно будет рассматриваться в разделе 19.8.1. Однако, прежде чем углубиться в особенности синтаксиса селекторов, мы сначала разберемся с тем, что можно делать с результатами запроса.

Возвращаемым значением функции `$()` является объект `jQuery`. Объекты `jQuery` – это объекты, подобные массивам: они имеют свойство `length` и свойства с числовыми именами, начиная с 0 до `length-1`. (Подробнее об объектах, подобных массивам, рассказывается в разделе 7.11.) Это означает, что к содержимому объекта `jQuery` можно обращаться, используя стандартный синтаксис обращения к элементам массива с квадратными скобками:

```
$("body").length // => 1: документ имеет единственный элемент body
$("body")[0]     // То же самое, что document.body
```

Если при работе с объектом `jQuery` вы предпочитаете не использовать синтаксис массивов, вместо свойства `length` можно использовать метод `size()`, а вместо индексов в квадратных скобках – метод `get()`. Если потребуется преобразовать объект `jQuery` в настоящий массив, можно вызвать метод `toArray()`.

В дополнение к свойству `length` объекты `jQuery` имеют еще три свойства, представляющие определенный интерес. Свойство `selector` хранит строку селектора (если таковая имеется), которая использовалась при создании объекта `jQuery`. Свойство `context` ссылается на объект контекста, который был передан функции `$()` во втором аргументе, в противном случае оно будет ссылаться на объект `Document`. Наконец, все объекты `jQuery` имеют свойство `jquery`, проверка наличия которого является самым простым способом отличить объект `jQuery` от любого другого объекта, подобного массиву. Значением свойства `jquery` является строка с номером версии библиотеки jQuery:

```
// Отыскать все элементы <script> в теле документа
var bodyscripts = $("script", document.body);
bodyscripts.selector // => "script"
bodyscripts.context  // => document.body
bodyscripts.jquery   // => "1.4.2"
```

Если потребуется обойти в цикле все элементы в объекте `jQuery`, вместо цикла `for` можно использовать метод `each()`. Метод `each()` напоминает метод `forEach()` массивов, определяемый стандартом ECMAScript 5 (ES5). В единственном аргументе он принимает функцию обратного вызова, которая будет вызвана для каждого элемента в объекте `jQuery` (в порядке следования в документе). Эта функция вызывается как метод элемента, т. е. внутри функции ключевое слово `this` ссылается на объект `Element`. Кроме того, метод `each()` передает функции обратного вызова индекс и элемент в первом и втором аргументах. Обратите внимание, что ссылка `this` и второй аргумент ссылаются на обычные элементы документа, а не на объекты `jQuery`. Если для работы с элементом внутри функции потребуется использовать методы объекта `jQuery`, передайте этот элемент функции `$()`.

Метод `each()` объекта `jQuery` имеет одну особенность, которая существенно отличает его от метода `forEach()`: если функция обратного вызова вернет `false` для какого-либо элемента, итерации будут остановлены после этого элемента (это напоминает использование ключевого слова `break` в обычном цикле). Метод `each()` возвращает объект `jQuery`, относительно которого он был вызван, благодаря чему он может использоваться в цепочках вызовов методов. Например (здесь используется метод `prepend()`, который будет описан в разделе 19.3):

```
// Пронумеровать элементы div документа вплоть до элемента div#last (включительно)
$("div").each(function(idx) { // отыскать все элементы <div> и обойти их
    $(this).prepend(idx + ": ");          // Вставить индекс в начало каждого
    if (this.id === "last") return false; // Остановиться по достижении
});                                       // элемента #last
```

Несмотря на широту возможностей, метод `each()` не слишком часто используется на практике, поскольку методы объекта `jQuery` обычно неявно выполняют итерации по всем выбранным элементам и выполняют операции над ними всеми. Надобность в методе `each()` обычно возникает только в случае, когда необходимо обработать выбранные элементы каким-то другим способом. Но даже в этом случае необходимость в методе `each()` может отсутствовать, поскольку многие методы объекта `jQuery` позволяют передавать функцию обратного вызова.

Библиотека jQuery поддерживает методы массивов, определяемые стандартом ES5, и содержит пару методов, по своей функциональности похожих на методы в стандарте ES5. Метод `map()` объекта `jQuery` действует практически так же, как метод `Array.map()`. Он принимает функцию обратного вызова в виде аргумента и вызывает ее для каждого элемента в объекте `jQuery`, собирая значения, возвращаемые этой функцией, и возвращая новый объект `jQuery`, хранящий эти значения. Метод `map()` вызывает функцию точно так же, как это делает метод `each()`: он передает ей элемент в виде ссылки `this` и во втором аргументе, а в первом аргументе – индекс элемента. Если функция обратного вызова вернет `null` или `undefined`, это значение будет проигнорировано и не будет добавлено в новый объект `jQuery`. Если функция обратного вызова вернет массив или объект, подобный массиву (такой как объект `jQuery`), этот объект будет «развернут» и содержащиеся в нем элементы по отдельности будут добавлены в новый объект `jQuery`. Обратите внимание, что объект `jQuery`, возвращаемый методом `map()`, может хранить объекты, не являющиеся элементами документа, но он по-прежнему будет действовать как объект, подобный массиву. Например:

```
// Отыскать все заголовки, отобразить их в значения их атрибутов id,
// преобразовать результат в настоящий массив и отсортировать его.
$(":header").map(function() { return this.id; }).toArray().sort();
```

$() в сравнении с querySelectorAll()

Функция `$()` похожа на метод `querySelectorAll()` объекта `Document`, с которым мы познакомились в разделе 15.2.5: оба принимают CSS-селектор в виде аргумента и возвращают объект, подобный массиву, хранящий элементы, соответствующие селектору. Библиотека jQuery использует метод `querySelectorAll()` в броузерах, поддерживающих его, однако существуют веские причины, почему в своих программах следует использовать функцию `$()`, а не метод `querySelectorAll()`:

- Метод `querySelectorAll()` был реализован производителям броузеров относительно недавно. Функция `$()` работает не только в новых, но и в старых броузерах.
- Благодаря тому что библиотека jQuery может производить выборку элементов «вручную», селекторы CSS3, поддерживаемые функцией `$()`, могут использоваться во всех броузерах, а не только в тех, что поддерживают CSS3.
- Объект, подобный массиву, возвращаемый функцией `$()` (объект `jQuery`), намного удобнее в работе, чем объект (`NodeList`), возвращаемый методом `querySelectorAll()`.

Наряду с методами `each()` и `map()` объект `jQuery` имеет еще один фундаментальный метод – `index()`. Этот метод принимает элемент в виде аргумента и возвращает его индекс в объекте `jQuery` или -1, если указанный элемент не будет найден. Однако, что типично для `jQuery`, метод `index()` имеет перегруженные версии. Если в качестве аргумента передать методу `index()` объект `jQuery`, он попытается отыскать первый элемент из этого объекта. Если передать строку, метод `index()` будет использовать ее как CSS-селектор и вернет индекс первого элемента в объекте `jQuery`, соответствующего селектору. А если вызвать метод `index()` без аргументов, он вернет индекс первого элемента в объекте `jQuery` среди элементов одного с ним уровня вложенности.

Последним методом общего назначения объекта `jQuery`, с которым мы познакомимся здесь, является метод `is()`. Он принимает селектор в виде аргумента и возвращает `true`, если хотя бы один из выбранных элементов соответствует указан-

ному селектору. Его можно использовать в функции обратного вызова, передаваемой методу `each()`**, например:**

```
$("div").each(function() {            // Для каждого элемента <div>
   if ($(this).is(":hidden")) return; // Пропустить скрытые элементы
   // Выполнить операции с видимыми элементами
});
```

19.2. Методы чтения и записи объекта jQuery

Простейшими и часто используемыми операциями, которые выполняются над объектами `jQuery`**, являются операции чтения и изменения значений HTML-атрибутов, стилей CSS, содержимого элементов и их геометрии. В данном разделе описываются методы, используемые для выполнения этих операций. Однако перед этим необходимо сделать некоторые обобщения, касающиеся методов чтения и записи объекта jQuery:**

- **Вместо того чтобы определять парные методы, библиотека jQuery использует одни и те же методы как для чтения, так и для записи. Если передать методу новое значение, он запишет это значение. Если новое значение не указано, метод вернет текущее значение.**
- **При использовании методов для записи они записывают новое значение во все элементы, находящиеся в объекте** `jQuery`**, и возвращают объект** `jQuery`**, что позволяет использовать их в цепочках вызовов методов.**
- **При использовании методов для чтения они читают значение только из первого элемента в наборе и возвращают единственное значение. (Если необходимо получить значения из всех элементов, используйте метод** `map()`**.) Поскольку методы чтения не возвращают объект** `jQuery`**, они могут использоваться только в конце цепочек вызовов методов.**
- **При использовании методов для записи они часто могут принимать объекты в виде аргументов. В этом случае каждое свойство указанного объекта будет определять имя и устанавливаемое значение.**
- **При использовании методов для записи они часто могут принимать вместо значений функции. В этом случае функция будет использоваться для вычисления устанавливаемого значения. Элемент, для которого должно быть вычислено значение, будет передан функции в ссылке** `this`**, в первом аргументе будет передан индекс элемента, а во втором аргументе – текущее значение.**

Продолжая чтение раздела, помните об этих общих чертах методов чтения и записи. Каждый из следующих подразделов описывает отдельную категорию методов чтения/записи объекта `jQuery`**.**

19.2.1. Чтение и запись значений HTML-атрибутов

Метод `attr()` **объекта** `jQuery` **– это метод чтения/записи значений HTML-атрибутов, и к нему относятся все обобщения, описанные выше. Метод** `attr()` **предусматривает решение проблемы несовместимости броузеров и обработку специальных случаев и позволяет использовать имена HTML-атрибутов или имена эквивалентных им свойств в языке JavaScript (где они отличаются). Например, можно использовать имя «for» или «htmlFor», «class» или «className».** `removeAttr()` **– это**

родственная функция, которая полностью удаляет атрибут из всех выбранных элементов. Ниже приводятся несколько примеров использования этих методов:

```
$("form").attr("action");                 // Получить атрибут action 1-й формы
$("#icon").attr("src", "icon.gif");       // Установить атрибут src
$("#banner").attr({src:"banner.gif", alt:"Advertisement", // Установить сразу 4 атрибута
                   width:720, height:64});
$("a").attr("target", "_blank");          // Все ссылки загружать в новых окнах
$("a").attr("target", function() {        // Локальные ссылки загружать локально
    if (this.host == location.host) return "_self"
    else return "_blank";                 // Внешние ссылки загружать в новых окнах
});
$("a").attr({target: function() {...}}); // Можно также передать функцию
$("a").removeAttr("target");              // Все ссылки загружать в этом окне
```

19.2.2. Чтение и запись значений CSS-атрибутов

Метод `css()` напоминает метод `attr()`, но работает не с HTML-атрибутами, а со стилями CSS элемента. При чтении значений стилей метод `css()` возвращает текущий (или «вычисленный»; раздел 16.4) стиль элемента: возвращаемое значение может быть определено в атрибуте `style` или в таблице стилей. Обратите внимание, что нельзя получить значения составных стилей, таких как «font» или «margin». Вместо этого следует запрашивать отдельные стили, такие как «font-weight», «font-family», «margin-top» или «margin-left». При записи значений стилей метод `css()` просто добавляет их в атрибут `style` элемента. Метод `css()` позволяет указывать имена стилей CSS с дефисами («background-color») или имена свойств в языке JavaScript с переменным регистром символов («backgroundColor»). При чтении значений стилей метод `css()` возвращает числовые значения в виде строк с добавлением единиц измерения в конце. Однако при записи он преобразует числа в строки и добавляет суффикс «px» (pixels – пикселы), если это необходимо:

```
$("h1").css("font-weight"); // Насыщенность шрифта первого элемента <h1>
$("h1").css("fontWeight");  // Допускается использовать имена свойств
$("h1").css("font");        // Ошибка: нельзя запрашивать составные стили
$("h1").css("font-variant", "smallcaps");        // Установить стиль всех элементов <h1>
$("div.note").css("border", "solid black 2px"); // Составные стили можно устанавливать
$("h1").css({ backgroundColor: "black",      // Записать сразу несколько стилей
              textColor: "white",            // имена с переменным регистром
              fontVariant: "small-caps",     // лучше подходят на роль имен
              padding: "10px 2px 4px 20px",  // свойств объекта
              border: "dotted black 4px" });
// Увеличить размер шрифта во всех элементах <h1> на 25%
$("h1").css("font-size", function(i,curval) {
    return Math.round(1.25*parseInt(curval));
});
```

19.2.3. Чтение и запись CSS-классов

Напомню, что значение атрибута `class` (в языке JavaScript доступного в виде свойства `className`) интерпретируется как список имен классов CSS, разделенных пробелами. Обычно бывает необходимо добавить, удалить или проверить присутствие какого-то одного имени в списке, тогда как потребность замещать один список классов другим возникает крайне редко. По этой причине в объекте `jQuery` определены удобные методы для работы с атрибутом `class`. Методы `addClass()` и `removeClass()` добавляют и удаляют классы в выбранных элементах. Метод `toggleClass()` добавляет классы в элемент, если они отсутствуют в нем, и удаляет их, если они присутствуют. Метод `hasClass()` проверяет присутствие указанного класса. Ниже приводятся несколько примеров использования этих методов:

```
// Добавление CSS-классов
$("h1").addClass("hilite");          // Добавить класс во все элементы <h1>
$("h1+p").addClass("hilite first");  // Добавить 2 класса в <p> после <h1>
$("section").addClass(function(n) {  // Передать функцию, чтобы добавить
    return "section" + n;            // вычисляемый класс во все
});                                  // выбранные элементы

// Удаление CSS-классов
$("p").removeClass("hilite");        // Удалить класс из всех элементов <p>
$("p").removeClass("hilite first");  // Допустимо удалять несколько классов
$("section").removeClass(function(n) { // Удалить вычисляемый класс из элем.
    return "section" + n;
});
$("div").removeClass();              // Удалить все классы из всех <div>

// Переключение CSS-классов
$("tr:odd").toggleClass("oddrow");   // Добавить класс, если отсутствует,
                                     // или удалить в противном случае
$("h1").toggleClass("big bold");     // Переключить сразу два класса
$("h1").toggleClass(function(n) {    // Переключить вычисляемый класс
    return "big bold h1-" + n;       // или классы
});
$("h1").toggleClass("hilite", true); // Действует как addClass
$("h1").toggleClass("hilite", false); // Действует как removeClass

// Проверка CSS-классов
$("p").hasClass("first")             // Имеет ли какой-нибудь <p> этот класс?
$("#lead").is(".first")              // То же самое
$("#lead").is(".first.hilite")       // is() - более гибкий, чем hasClass()
```

Обратите внимание, что метод `hasClass()` не такой гибкий, как методы `addClass()`, `removeClass()` и `toggleClass()`. Метод `hasClass()` может работать только с одним именем класса и не поддерживает возможность передачи ему функции. Он возвращает `true`, если хотя бы один из выбранных элементов содержит указанный класс, и `false` – в противном случае. Метод `is()` (описываемый в разделе 19.1.2) более гибкий и может использоваться для той же цели.

Эти методы объекта `jQuery` подобны методам свойства `classList`, о котором рассказывалось в разделе 16.5, но методы объекта `jQuery` работают во всех броузерах, а не только в тех, которые поддерживают свойство `classList`, определяемое стандартом HTML5. И, конечно же, методы объекта `jQuery` работают с множеством элементов и могут добавляться в цепочки вызовов методов.

19.2.4. Чтение и запись значений элементов HTML-форм

Метод `val()` служит для чтения и записи значений атрибутов `value` элементов HTML-форм, а также для чтения и записи состояния выбора флажков, радиокнопок и элементов `<select>`:

```
$("#surname").val()      // Получить значение текстового поля surname
$("#usstate").val()      // Получить единственное значение из элемента <select>
$("select#extras").val() // Получить массив значений из <select multiple>
$("input:radio[name=ship]:checked").val()  // Получить значение атрибута
                                           // checked радиокнопки
$("#email").val("Invalid email address")   // Установить значение текст. поля
$("input:checkbox").val(["opt1", "opt2"])  // Установить флажки с указанными именами
                                           // или значениями
$("input:text").val(function() {           // Сбросить все текстовые поля
    return this.defaultValue;              // в значения по умолчанию
})
```

19.2.5. Чтение и запись содержимого элемента

Методы `text()` и `html()` читают и записывают содержимое элемента или элементов в виде простого текста или разметки HTML. При вызове без аргументов метод `text()` возвращает содержимое всех вложенных текстовых узлов из всех выбранных элементов в виде простого текста. Этот метод работает даже в броузерах, не поддерживающих свойства `textContent` и `innerText` (раздел 15.5.2).

Если вызвать метод `html()` без аргументов, он вернет в виде разметки HTML содержимое только первого выбранного элемента. Для этой цели библиотека jQuery использует свойство `innerHTML`: `x.html()` – фактически то же самое, что и `x[0].innerHTML`.

Если методу `text()` или `html()` передать строку, она будет использована как содержимое элемента в виде простого текста или разметки HTML и заместит текущее его содержимое. Подобно другим методам записи, с которыми мы уже познакомились, этим методам можно также передать функцию, которая будет использована для получения строки с новым содержимым:

```
var title = $("head title").text()       // Получить заголовок документа
var headline = $("h1").html()            // Получить разметку html первого <h1>
$("h1").text(function(n,current) {       // Добавить в каждый заголовок
    return "§" + (n+1) + ": " + current  // порядковый номер раздела
});
```

19.2.6. Чтение и запись параметров геометрии элемента

В разделе 15.8 мы узнали, насколько сложно бывает определить размер и координаты элемента, особенно в броузерах, не поддерживающих метод `getBoundingClientRect()` (раздел 15.8.2). Библиотека jQuery упрощает эти вычисления, предоставляя методы, работающие в любых броузерах. Обратите внимание, что все методы, описываемые здесь, являются методами чтения, и лишь некоторые из них могут использоваться для записи.

Метод `offset()` позволяет получить или изменить координаты элемента. Этот метод определяет координаты относительно начала документа и возвращает их в виде объекта со свойствами `left` и `top`, в которых хранятся координаты X и Y. Если передать методу объект с этими свойствами, он изменит координаты элемента в соответствии с указанными значениями. При необходимости он также установит CSS-атрибут `position`, чтобы сделать элемент позиционируемым:

```
var elt = $("#sprite");          // Элемент, который требуется переместить
var position = elt.offset();  // Получить текущие координаты
```

```
position.top += 100;         // Изменить координату Y
elt.offset(position);        // Переместить элемент в новую позицию

// Переместить все элементы <h1> вправо на расстояние, зависящее от их
// положения в документе
$("h1").offset(function(index,curpos) {
    return {left: curpos.left + 25*index, top:curpos.top};
});
```

Метод `position()` похож на метод `offset()`, за исключением того, что он позволяет только читать координаты и возвращает координаты элемента не относительно начала документа, а относительно его родителя. В разделе 15.8.5 мы узнали, что каждый элемент имеет свойство `offsetParent`, ссылающееся на родительский элемент, относительно которого определяются координаты. Позиционируемые элементы всегда играют роль начала координат для своих потомков, но некоторые броузеры дают эту роль и некоторым другим элементам, таким как ячейки таблицы. Роль начала координат в библиотеке jQuery могут играть только позиционируемые элементы, и метод `offsetParent()` объекта `jQuery` отображает каждый элемент на ближайший позиционируемый вмещающий элемент или на элемент `<body>`. Следует отметить не совсем удачный выбор имен этих методов: `offset()` возвращает абсолютные координаты элемента относительно начала документа, а метод `position()` возвращает смещение элемента относительно его ближайшего предка `offsetParent()`.

Существует также три метода чтения, позволяющие получить ширину, и три метода чтения, позволяющие получить высоту элемента. Методы `width()` и `height()` возвращают базовые значения ширины и высоты, не включающие отступы, рамки и поля. Методы `innerWidth()` и `innerHeight()` возвращают ширину и высоту элемента с отступами (слово «inner» (внутренний) указывает на тот факт, что эти методы возвращают ширину и высоту внутри рамки). Методы `outerWidth()` и `outerHeight()` по умолчанию возвращают размеры элемента с отступами и рамкой. Если этим методам передать значение `true`, они добавят размеры полей элемента. Следующий фрагмент демонстрирует, что для элемента можно получить четыре разные ширины:

```
var body = $("body");
var contentWidth = body.width();
var paddingWidth = body.innerWidth();
var borderWidth = body.outerWidth();
var marginWidth = body.outerWidth(true);
var padding = paddingWidth-contentWidth; // сумма левого и правого отступов
var borders = borderWidth-paddingWidth;  // сумма толщины левой и правой рамки
var margins = marginWidth-borderWidth;   // сумма левого и правого полей
```

Методы `width()` и `height()` обладают свойствами, отсутствующими у других четырех методов (методов `inner` и `outer`). Во-первых, если первый элемент в объекте `jQuery` является объектом `Window` или `Document`, эти методы вернут размер видимой области окна или полный размер документа. Другие методы работают только с элементами, не являющимися окнами или документами.

Другая особенность методов `width()` и `height()` заключается в том, что они являются также методами записи. Если передать значение этим методам, они установят ширину или высоту всех элементов в объекте `jQuery`. (Обратите, однако, внимание, что они не могут изменять ширину или высоту объектов `Window` и `Document`.) Если аргумент имеет числовое значение, он будет интерпретироваться как число

пикселов. Если передать строку, она будет использоваться как значение CSS-атрибута `width` или `height`, благодаря чему в ней можно указывать любые единицы измерения, предусматриваемые стандартом CSS. Наконец, как и другие методы записи, они могут принимать функцию, которая будет вызываться для вычисления значения ширины или высоты.

Между ролями чтения и записи методов `width()` и `height()` существует маленькое несоответствие. При использовании в качестве методов чтения они возвращают размеры элемента по содержимому, исключая отступы, рамки и поля. Однако при использовании в качестве методов записи они просто устанавливают CSS-атрибуты `width` и `height`. По умолчанию эти атрибуты также определяют размеры по содержимому. Но если элемент имеет CSS-атрибут `box-sizing` (раздел 16.2.3.1), установленный в значение `border-box`, методы `width()` и `height()` будут устанавливать размеры, включающие ширину отступов и рамок. Для элемента `e`, использующего блочную модель «content-box», вызов `$(e).width(x).width()` вернет значение `x`. Однако для элемента, использующего блочную модель «border-box», этот же вызов в общем случае вернет другое значение.

Последняя пара методов объекта `jQuery`, имеющих отношение к геометрии элементов, – это методы `scrollTop()` и `scrollLeft()`, позволяющие получить позиции полос прокрутки для элемента или множество позиций полос прокрутки для всех элементов. Эти методы могут применяться и к элементам документа, и к объекту `Window`, а при вызове для объекта `Document` они возвращают или устанавливают позиции полос прокрутки объекта `Window`, хранящего документ. В отличие от других методов записи, методам `scrollTop()` и `scrollLeft()` нельзя передавать функции.

Метод `scrollTop()` как метод чтения и записи можно использовать в паре с методом `height()`, чтобы на их основе определить метод, прокручивающий окно на указанное число страниц:

```
// Прокручивает окно на n страниц. n может быть дробным и отрицательным числом
function page(n) {
    var w = $(window);                  // Обернуть окно объектом jQuery
    var pagesize = w.height();          // Получить размер страницы
    var current = w.scrollTop();        // Текущие позиции полос прокрутки
    w.scrollTop(current + n*pagesize);  // Установить новые позиции
}                                       // полос прокрутки
```

19.2.7. Чтение и запись данных в элементе

Библиотека jQuery определяет метод чтения/записи с именем `data()`, который возвращает или устанавливает данные, связанные с любым элементом документа или с объектами `Document` и `Window`. Возможность связывать данные с любыми элементами является одной из наиболее важных и мощных особенностей: она лежит в основе механизма регистрации обработчиков событий и последовательностей визуальных эффектов в библиотеке jQuery, и в определенных случаях метод `data()` может оказаться полезным в ваших сценариях.

Чтобы связать данные с элементами в объекте `jQuery`, нужно вызвать `data()` как метод записи, передав ему в виде двух аргументов имя и значение. Методу `data()` как методу записи можно также передать единственный объект, каждое свойство которого будет использоваться как пара имя/значение, связываемая с элементом или элементами в объекте `jQuery`. Однако обратите внимание, что, когда методу data() передается объект, свойства этого объекта будут замещать все данные, ранее связанные с элементом или элементами. В отличие от многих других методов записи, с которыми мы уже познакомились, метод data() не вызывает функцию, переданную ему. Если во втором аргументе передать методу data() функцию, она будет сохранена, как любое другое значение.

Конечно, метод data() может также использоваться в роли метода чтения. При вызове без аргументов он возвращает объект, содержащий все пары имя/значение, связанные с первым элементом в объекте jQuery. При вызове метода data() с единственным строковым аргументом он возвращает значение, связанное с этой строкой в первом элементе.

Для удаления данных из элемента или элементов можно использовать метод removeData(). (Вызов метода data() с именованным значением null или undefined фактически не удаляет данные.) Если методу removeData() передать строку, он удалит значение, связанное с этой строкой в элементе или элементах. Если вызвать метод removeData() без аргументов, он удалит все данные:

```
$("div").data("x", 1);              // Записать некоторые данные
$("div.nodata").removeData("x");  // Удалить некоторые данные
var x = $('#mydiv').data("x");      // Получить некоторые данные
```

Кроме того, библиотека jQuery определяет вспомогательные функции, действующие аналогично методам data() и removeData(). Таким образом, связать данные с отдельным элементом e можно с помощью метода или функции data():

```
$(e).data(...) // Метод
$.data(e, ...) // Функция
```

Механизм хранения данных в библиотеке jQuery не использует для этой цели свойства самих элементов, но добавляет одно специальное свойство ко всем элементам, имеющим связанные с ними данные. Некоторые броузеры не позволяют добавлять свойства к элементам <applet>, <object> и <embed>, поэтому библиотека jQuery просто не дает возможности связать данные с элементами этих типов.

19.3. Изменение структуры документа

В разделе 19.2.5 мы познакомились с методами html() и text(), позволяющими изменять содержимое элемента. В этом разделе будут рассматриваться методы, позволяющие производить более сложные изменения в документе. В броузере HTML-документы представлены в виде дерева узлов, а не в виде линейной последовательности символов, поэтому вставку, удаление и замену фрагмента документа выполнить не так просто, как фрагмента строки или массива. В следующих подразделах описываются различные методы объекта jQuery, предназначенные для внесения изменений в документ.

19.3.1. Вставка и замена элементов

Начнем с самых основных методов вставки и замены. Все методы, демонстрирующиеся ниже, принимают аргумент, определяющий содержимое, которое должно быть вставлено в документ. Это может быть строка с простым текстом или с разметкой HTML, определяющая содержимое, объект jQuery, элемент Element или текстовый узел Node. Вставка может быть выполнена внутрь, перед, после или

вместо (в зависимости от метода) каждого выбранного элемента. Если в качестве вставляемого содержимого используется элемент, уже присутствующий в документе, он перемещается из текущего местоположения. Если выполняется вставка сразу в несколько мест, элемент будет скопирован необходимое число раз. Все эти методы возвращают объект `jQuery`, относительно которого они вызываются. Обратите, однако, внимание, что после вызова метода `replaceWith()` элементы, находящиеся в объекте `jQuery`, исключаются из документа:

```
$("#log").append("<br/>"+message); // Добавить содержимое в конец элем. #log
$("h1").prepend("§");        // Добавить символ параграфа в начало каждого элемента <h1>
$("h1").before("<hr/>");           // Вставить линию перед каждым элем. <h1>
$("h1").after("<hr/>");            // И после
$("hr").replaceWith("<br/>");      // Заменить элементы <hr/> на <br/>
$("h2").each(function() {          // Заменить <h2> на <h1>,
               var h2 = $(this);   // сохранив содержимое
               h2.replaceWith("<h1>" + h2.html() + "</h1>");
           });
// Методы after() и before() могут также применяться к текстовым узлам
// Ниже демонстрируется другой способ добавления символа параграфа во все <h1>
$("h1").map(function() { return this.firstChild; }).before("§");
```

Каждый из этих пяти методов, изменяющих структуру документа, может также принимать функцию, которая должна вызываться для вычисления вставляемого значения. Как обычно, в этом случае функция будет вызываться по одному разу для каждого выбранного элемента. Ссылка `this` в ней будет указывать на текущий элемент, а в первом аргументе ей будет передаваться индекс элемента в объекте `jQuery`. Методы `append()`, `prepend()` и `replaceWith()` будут передавать функции во втором аргументе текущее содержимое элемента в виде строки с разметкой HTML. А методы `before()` и `after()` будут вызывать функцию без второго аргумента.

Все пять методов, представленные выше, применяются к целевым элементам и принимают вставляемое содержимое в виде аргумента. Для каждого из этих пяти методов имеется парный метод, действующий в обратном порядке: он вызывается относительно содержимого и принимает целевые элементы в виде аргументов. Пары методов перечислены в следующей таблице:

Операция	$(элементы).*метод*(содержимое)	$(содержимое).*метод*(элементы)
вставка содержимого в конец целевого элемента	`append()`	`appendTo()`
вставка содержимого в начало целевого элемента	`prepend()`	`prependTo()`
вставка содержимого после целевого элемента	`after()`	`insertAfter()`
вставка содержимого перед целевым элементом	`before()`	`insertBefore()`
замена целевого элемента содержимым	`replaceWith()`	`replaceAll()`

Методы, продемонстрированные в примере выше, перечислены во втором столбце. Методы, перечисленные в третьем столбце, будут демонстрироваться ниже. Есть несколько важных замечаний, касающихся этих пар методов:

- Если любому методу из второго столбца передать строку, он будет интерпретировать ее как разметку HTML. Если передать строку любому методу из третьего столбца, он будет интерпретировать ее как селектор, идентифицирующий целевые элементы. (Целевые элементы можно также указывать явно, передавая объект `jQuery`, `Element` или `Node`.)
- Методы из третьего столбца не принимают функции в аргументах, в отличие от методов из второго столбца.
- Методы из второго столбца возвращают объект `jQuery`, относительно которого они вызываются. Элементы в этом объекте `jQuery` могут иметь новое содержимое или новые братские элементы, но сами они не изменяются. Методы из третьего столбца вызываются относительно содержимого, которое должно быть вставлено, и возвращают новый объект `jQuery`, представляющий новое содержимое после вставки. В частности, обратите внимание, что если содержимое вставляется сразу в несколько мест, возвращаемый объект `jQuery` будет включать по одному элементу для каждой позиции вставки.

После перечисления различий реализуем те же операции, что и в примере выше, с помощью методов из третьего столбца. Обратите внимание, что во второй строке методу `$()` нельзя передать простой текст (без угловых скобок, которые позволили бы идентифицировать его как разметку HTML), потому что он будет интерпретироваться как селектор. По этой причине требуется явно создать текстовый узел, который должен быть вставлен:

```
$("<br/>+message").appendTo("#log"); // Добавить разметку html в #log
$(document.createTextNode("§")).prependTo("h1"); // Добавить текстовый узел во все <h1>
$("<hr/>").insertBefore("h1");       // Вставить линию перед каждым <h1>
$("<hr/>").insertAfter("h1");        // Вставить линию после каждого <h1>
$("<br/>").replaceAll("hr");         // Заменить элементы <hr/> на <br/>
```

19.3.2. Копирование элементов

Как отмечалось выше, при вставке элементов, уже являющихся частью документа, эти элементы не копируются, а просто перемещаются в новое местоположение. Если элемент вставляется в несколько мест, библиотека jQuery скопирует элемент столько раз, сколько потребуется, но копирование не выполняется при вставке только в одно местоположение. Если потребуется не переместить, а скопировать элемент, необходимо сначала создать копию с помощью метода `clone()`. Метод `clone()` создает и возвращает копии всех выбранных элементов (и всех потомков этих элементов). Копии элементов, находящиеся в возвращаемом объекте `jQuery`, не являются частью документа, но их можно вставить в документ с помощью любого из методов, представленных выше:

```
// Добавить новый div с атрибутом id="linklist" в конец документа
$(document.body).append("<div id='linklist'><h1>List of Links</h1></div>");
// Скопировать все ссылки в документе и вставить их в этот новый div
$("a").clone().appendTo("#linklist");
// Вставить элементы <br/> после каждой ссылки, чтобы они отображались в отдельных строках
$("#linklist > a").after("<br/>");
```

По умолчанию метод `clone()` не копирует обработчики событий (раздел 19.4) и другие данные (раздел 19.2.7), связанные с элементами. Если необходимо будет скопировать эти дополнительные данные, передайте методу `clone()` значение `true`.

19.3.3. Обертывание элементов

Другой способ вставки элементов в HTML-документ связан с обертыванием новым элементом (или элементами) одного или более элементов. Объект `jQuery` определяет три метода обертывания. Метод `wrap()` обертывает каждый выбранный элемент. Метод `wrapInner()` обертывает содержимое каждого выбранного элемента. А метод `wrapAll()` обертывает все выбранные элементы как единое целое. По умолчанию этим методам передается вновь созданный обертывающий элемент или строка с разметкой HTML, которая будет использована для создания обертки. Строка с разметкой HTML может включать вложенные элементы, если это необходимо, но на самом верхнем уровне она должна содержать единственный элемент. Если любому из этих методов передать функцию, она будет вызываться по одному разу в контексте каждого элемента (с индексом элемента в виде единственного аргумента) и должна возвращать строку, элемент `Element` или объект `jQuery`. Например:

```
// Обернуть каждый элемент <h1> элементом <i>
$("h1").wrap(document.createElement("i")); // Результат: <i><h1>...</h1></i>
// Обернуть содержимое каждого элемента <h1>. Строковый аргумент проще в использовании.
$("h1").wrapInner("<i/>");                 // Результат: <h1><i>...</i></h1>
// Обернуть первый абзац якорем и элементом div
$("body>p:first").wrap("<a name='lead'><div class='first'></div></a>");
// Обернуть все остальные абзацы другим элементом div
$("body>p:not(:first)").wrapAll("<div class='rest'></div>");
```

19.3.4. Удаление элементов

Помимо методов вставки и замены в объекте `jQuery` имеются также методы удаления элементов. Метод `empty()` удаляет все дочерние элементы (включая текстовые узлы) из каждого выбранного элемента без изменения самого элемента. Метод `remove()`, напротив, удаляет из документа все выбранные элементы (и все их содержимое). Обычно метод `remove()` вызывается без аргументов и удаляет все элементы, находящиеся в объекте `jQuery`. Однако если передать методу аргумент, этот аргумент будет интерпретироваться как селектор, и удалены будут только элементы из объекта `jQuery`, соответствующие селектору. (Если необходимо удалить элементы из множества выбранных элементов, не удаляя их из документа, используйте метод `filter()`, о котором рассказывается в разделе 19.8.2.) Обратите внимание, что не требуется удалять элементы перед повторной их вставкой в документ: достаточно просто вставить их в новое местоположение, а библиотека автоматически переместит их.

Метод `remove()` удаляет также все обработчики событий (раздел 19.4) и другие данные (раздел 19.2.7), которые могли быть связаны с удаляемыми элементами. Метод `detach()` действует подобно методу `remove()`, но не удаляет обработчики событий и данные. Метод `detach()` может оказаться удобнее, когда элементы требуется удалить из документа на время и позднее вставить их обратно.

Наконец, метод `unwrap()` выполняет удаление элементов способом, противоположным тому, каким действует метод `wrap()` или `wrapAll()`: он удаляет родительский элемент каждого выбранного элемента, не оказывая влияния на выбранные элементы и их братские элементы. То есть для каждого выбранного элемента он замещает родителя этого элемента его дочерними элементами. В отличие от методов `remove()` и `detach()`, метод `unwrap()` не принимает необязательный аргумент с селектором.

19.4. Обработка событий с помощью библиотеки jQuery

Как мы узнали в главе 17, одна из сложностей, связанных с обработкой событий, состоит в том, что в IE (до версии IE9) реализована модель событий, отличающаяся от модели событий в других броузерах. Чтобы решить эту проблему, в библиотеке jQuery определяется собственная унифицированная модель событий, которая одинаково работает во всех броузерах. В простейших случаях модель jQuery API оказывается проще в использовании, чем стандартная модель или модель IE. В более сложных случаях модель jQuery предоставляет более широкие возможности, чем стандартная модель. Все дополнительные подробности описываются в следующих подразделах.

19.4.1. Простые методы регистрации обработчиков событий

Библиотека jQuery определяет простые методы регистрации обработчиков для всех наиболее часто используемых событий, поддерживаемых всеми броузерами. Например, чтобы зарегистрировать обработчик события «click», достаточно просто вызвать метод `click()`:

```
// Щелчок на любом элементе <p> окрашивает его фон в серый цвет
$("p").click(function() { $(this).css("background-color", "gray"); });
```

Вызов метода объекта `jQuery` регистрации обработчика событий зарегистрирует указанный обработчик во всех выбранных элементах. Обычно это проще, чем регистрировать один и тот же обработчик во всех элементах по отдельности с помощью метода `addEventListener()` или `attachEvent()`.

Библиотека jQuery определяет следующие простые методы регистрации обработчиков событий:

`blur()`
`change()`
`click()`
`dblclick()`
`error()`
`focus()`
`focusin()`
`focusout()`
`keydown()`
`keypress()`
`keyup()`
`load()`
`mousedown()`
`mouseenter()`
`mouseleave()`
`mousemove()`
`mouseout()`
`mouseover()`
`mouseup()`
`resize()`
`scroll()`
`select()`
`submit()`
`unload()`

Большая часть этих методов регистрации обработчиков наиболее часто используемых типов событий уже знакомы вам по главе 17. Тем не менее необходимо сделать несколько замечаний. События «focus» и «blur» не всплывают, в отличие от всплывающих событий «focusin» и «focusout», и библиотека jQuery гарантирует, что эти события будут работать во всех броузерах. События «mouseover» и «mouseout», наоборот, всплывают, и это часто доставляет неудобства, так как сложно определить – указатель мыши покинул интересующий нас элемент или событие было доставлено от одного из потомков. События «mouseenter» и «mouseleave» не всплывают, что решает данную проблему. Эти типы событий впервые появились в IE, но библиотека jQuery гарантирует, что они корректно будут работать во всех броузерах.

События «resize» и «unload» возбуждаются только в объекте `Window`, поэтому, если потребуется зарегистрировать обработчики этих типов событий, методы `resi-`

`ze()` и `unload()` следует вызывать относительно `$(window)`. Метод `scroll()` часто вызывается относительно `$(window)`, но его также можно вызывать относительно любых элементов, имеющих полосы прокрутки (например, относительно элементов, в которых CSS-атрибут `overflow` имеет значение «scroll» или «auto»). Метод `load()` может вызываться относительно `$(window)`, для регистрации обработчика события «load» окна, но обычно проще бывает передать свою функцию инициализации непосредственно функции `$()`, как показано в разделе 19.1.1. При этом метод `load()` с успехом можно использовать в элементах `<iframe>` и `<img>`. Обратите внимание, что при вызове метода `load()` с различными аргументами он также может использоваться для загрузки нового содержимого (по протоколу HTTP) в элемент (раздел 19.6.1). Метод `error()` можно использовать с элементами `<img>` для регистрации обработчиков неудачи загрузки изображения. Он не должен использоваться для установки свойства `onerror` объекта `Window`, описанного в разделе 14.6.

В дополнение к этим простым методам регистрации обработчиков событий существует две специальные формы методов, которые могут иногда пригодиться. Метод `hover()` регистрирует обработчики событий «mouseenter» и «mouseleave». Вызов `hover(f,g)` по своему действию аналогичен двум последовательным вызовам методов `mouseenter(f)` и `mouseleave(g)`. Если методу `hover()` передать единственный аргумент, он будет использоваться как обработчик обоих событий.

Другим специальным методом регистрации обработчиков событий является метод `toggle()`. Этот метод регистрирует функцию обработчика события «click». Вы можете указать две или более функции обработчиков, и библиотека jQuery будет вызывать их всякий раз, когда будет возникать событие «click». Если, например, вызвать этот метод как `toggle(f,g,h)`, функция `f()` будет вызываться для обработки первого события «click», `g()` – второго, `h()` – третьего и снова `f()` – для обработки четвертого события «click». Будьте внимательны при использовании метода `toggle()`: как будет показано в разделе 19.5.1, этот метод может также использоваться для отображения и скрытия (т. е. для переключения видимости) выбранных элементов.

В разделе 19.4.4 мы познакомимся с другими, более обобщенными способами регистрации обработчиков событий и завершим этот раздел описанием еще одного простого и удобного способа регистрации обработчиков.

Напомню, что создавать новые элементы можно с помощью функции `$()`, передавая ей строку с разметкой HTML и объект (во втором аргументе) с атрибутами, которые должны быть установлены во вновь созданном элементе. Второй аргумент может быть любым объектом, который допускается передавать методу `attr()`. Но кроме того, если имя какого-либо из свойств совпадает с именем метода регистрации обработчиков событий, перечисленных выше, значение этого свойства будет интерпретироваться как функция-обработчик и зарегистрировано как обработчик данного типа события. Например:

```
$("<img/>", {
    src: image_url,
    alt: image_description,
    className: "translucent_image",
    click: function() { $(this).css("opacity", "50%"); }
});
```

19.4.2. Обработчики событий в библиотеке jQuery

Функции обработчиков событий в примерах выше не имеют ни аргументов, ни возвращаемых значений. В целом это нормально для подобных обработчиков событий, но библиотека jQuery передает каждому обработчику событий один или более аргументов и анализирует значения, возвращаемые ими. Самое главное, что следует знать, – каждому обработчику событий библиотека jQuery передает в первом аргументе объект события. Поля этого объекта содержат дополнительную информацию о событии (такую как координаты указателя мыши). Свойства стандартного объекта Event были описаны в главе 17. Библиотека jQuery имитирует стандартный объект Event даже в броузерах (таких как IE версии 8 и ниже), не поддерживающих его, и объекты событий в библиотеке jQuery имеют одинаковые наборы полей во всех броузерах. Подробнее об этом рассказывается в разделе 19.4.3.

Обычно обработчикам событий передается единственный аргумент с объектом события. Но если событие генерируется явно с помощью метода trigger() (раздел 19.4.6), обработчикам можно передавать массив дополнительных аргументов. В этом случае дополнительные аргументы передаются обработчикам после первого аргумента с объектом события.

Независимо от способа регистрации значение, возвращаемое функцией обработчика событий, всегда имеет большое значение для библиотеки jQuery. Если обработчик вернет false, будут отменены и действия, предусмотренные по умолчанию для этого типа события, и дальнейшее распространение события. То есть возврат значения false равносилен вызову методов preventDefault() и stopPropagation() объекта Event. Кроме того, когда обработчик события возвращает значение (отличное от undefined), библиотека jQuery сохраняет это значение в свойстве result объекта Event, к которому можно обратиться в обработчиках событий, вызываемых вслед за этим обработчиком.

19.4.3. Объект Event в библиотеке jQuery

Библиотека jQuery скрывает различия в реализациях броузеров, определяя собственный объект Event. Когда библиотека jQuery вызывает обработчик события, она всегда передает ему в первом аргументе собственный объект Event. Объект Event в библиотеке jQuery основан на положениях стандартов консорциума W3C, но в нем также реализованы некоторые особенности, ставшие стандартными де-факто. jQuery копирует все следующие поля стандартного объекта Event во все свои объекты Event (хотя некоторые из них могут иметь значение undefined для определенных типов событий):

altKey
attrChange
attrName
bubbles
button
cancelable
charCode
clientX
clientY
ctrlKey
currentTarget
detail
eventPhase
fromElement
keyCode
layerX
layerY
metaKey
newValue
offsetX
offsetY
originalTarget
pageX
pageY
prevValue
relatedNode
relatedTarget
screenX
screenY
shiftKey
srcElement
target
toElement
view
wheelDelta
which

В дополнение к этим свойствам объект Event определяет также следующие методы:

```
preventDefault()              isDefaultPrevented()
stopPropagation()             isPropagationStopped()
stopImmediatePropagation()    isImmediatePropagationStopped()
```

Большинство из этих свойств и методов было представлено в главе 17 и описывается в четвертой части книги, в справочной статье `Event`. О некоторых из этих полей, обрабатываемых библиотекой jQuery особым образом, чтобы обеспечить им одинаковое поведение во всех броузерах, стоит упомянуть отдельно:

`metaKey`

Если стандартный объект события не имеет свойства `metaKey`, jQuery присваивает ему значение свойства `ctrlKey`. Значение свойства `metaKey` определяется в MacOS клавишей Command.

`pageX`, `pageY`

Если стандартный объект события не имеет этих свойств, но имеет свойства, определяющие координаты указателя мыши в видимой области в виде свойств `clientX` и `clientY`, jQuery вычислит координаты указателя мыши относительно начала документа и сохранит их в свойствах `pageX` и `pageY`.

`target`, `currentTarget`, `relatedTarget`

Свойство `target` ссылается на элемент документа, в котором возникло событие. Если это свойство в стандартном объекте события ссылается на текстовый узел, jQuery подставит ссылку на вмещающий объект `Element`. Свойство `currentTarget` ссылается на элемент, в котором был зарегистрирован текущий обработчик события. Значение этого свойства всегда должно совпадать со значением `this`.

Если значения свойств `currentTarget` и `target` не совпадают, следовательно, обрабатывается всплывшее событие и может оказаться полезным проверить элемент `target` с помощью метода `is()` (раздел 19.1.2):

```
if ($(event.target).is("a")) return; // Игнорировать события,
                                     // сгенерированные в ссылках
```

Свойство `relatedTarget` – ссылка на другой элемент, вовлеченный в события перехода, такие как «mouseover» и «mouseout». Например, для событий «mouseover» свойство `relatedTarget` будет ссылаться на элемент, который покинул указатель мыши при перемещении на элемент `target`. Если стандартный объект события не имеет свойства `relatedTarget`, но имеет свойства `toElement` и `fromElement`, свойство `relatedTarget` получает значение одного из этих свойств в зависимости от типа события.

`timeStamp`

Время возникновения события в миллисекундах, возвращаемое методом `Date.getTime()`. Библиотека jQuery сама устанавливает это свойство, чтобы обойти давнишнюю ошибку в Firefox.

`which`

Библиотека jQuery нормализует это нестандартное свойство события так, что оно определяет кнопку мыши или клавишу на клавиатуре, нажатие которой вызвало это событие. Для событий клавиатуры, если стандартный объект события не имеет свойства `which`, но имеет свойство `charCode` или `keyCode`, в свойство `which` будет записано значение свойства, которое определено. Для событий мыши, если свойство `which` отсутствует, но имеется свойство `button`, в свойство `which` будет записано значение, основанное на значении свойства `button`. 0 означает, что никакая кнопка не была нажата. 1 – была нажата левая кнопка, 2 – средняя кнопка и 3 – правая кнопка. (Обратите внимание, что в некоторых броузерах нажатие правой кнопки мыши не генерирует события.)

Кроме того, библиотека jQuery определяет следующие собственные поля в объекте `Event`, которые иногда могут оказаться полезными:

`data`

Если при регистрации обработчика события были указаны дополнительные данные (раздел 19.4.4), обработчик сможет получить к ним доступ с помощью этого поля.

`handler`

Ссылка на текущую функцию обработчика события.

`result`

Возвращаемое значение предыдущего обработчика этого события. Обработчики, не возвращающие ничего, не учитываются.

`originalEvent`

Ссылка на стандартный объект `Event`, созданный броузером.

19.4.4. Дополнительные способы регистрации обработчиков событий

Мы уже знаем, что в библиотеке jQuery имеется множество простых методов регистрации обработчиков событий. Внутри каждого из них вызывает один и тот же, более сложный метод `bind()`, который связывает обработчик с указанным типом события во всех элементах в объекте `jQuery`. Прямое использование метода `bind()` позволяет использовать дополнительные возможности механизма регистрации, недоступные в простых методах.[1]

В простейшем случае методу `bind()` передаются строка с типом события в первом аргументе и функция обработчика события во втором. Простые методы регистрации обработчиков событий используют именно эту форму вызова метода `bind()`. Вызов `$('p').click(f)`, например, эквивалентен вызову:

```
$('p').bind('click', f);
```

Метод `bind()` может также вызываться с тремя аргументами. В этой форме вызова тип события передается в первом аргументе, а функция обработчика – в третьем. Во втором аргументе можно передать любое значение, и библиотека jQuery будет присваивать это значение свойству `data` объекта `Event` перед вызовом обработчика. Иногда это может пригодиться для передачи обработчику дополнительных данных без использования замыкания.

---

[1] В библиотеке jQuery термин «связывание» используется для обозначения регистрации обработчиков событий. Стандарт ECMAScript 5 и многие фреймворки на языке JavaScript определяют в объектах функций метод `bind()` (раздел 8.7.4) и используют этот термин для обозначения связи функций с объектами, относительно которых они вызываются. Версия метода `Function.bind()` в библиотеке jQuery является вспомогательной функцией с именем `jQuery.proxy()`, которая описывается в разделе 19.7.

Кроме того, метод `bind()` обладает еще одной дополнительной особенностью. Если в первом аргументе передать список типов событий, разделенных пробелами, то функция обработчика будет зарегистрирована для всех указанных типов событий. Вызов `$('a').hover(f)` (раздел 19.4.1), например, эквивалентен вызову:

```
$('a').bind('mouseenter mouseleave', f);
```

Еще одной важной особенностью метода `bind()` является возможность указать при регистрации пространство (или пространства) имен для обработчиков событий. Это дает возможность определить группу обработчиков событий, что может пригодиться, когда позднее потребуется переключать или удалять обработчики из определенного пространства имен. Поддержка пространств имен для обработчиков особенно удобна для тех, кто пишет библиотеки или модули, использующие библиотеку jQuery. Пространства имен событий подобны селекторам CSS-классов. Чтобы связать обработчик события с определенным пространством имен, добавьте точку и имя пространства имен после типа события:

```
// Связать f, как обработчик события mouseover в пространстве имен "myMod"
// ко всем элементам <a>
$('a').bind('mouseover.myMod', f);
```

Имеется даже возможность связать обработчик с несколькими пространствами имен, например:

```
// Связать f, как обработчик события mouseout в пространствах имен "myMod" и "yourMod"
$('a').bind('mouseout.myMod.yourMod', f);
```

Последней особенностью метода `bind()` является возможность передать ему в первом аргументе объект, отображающий имена событий в функции обработчиков. Возьмем еще раз в качестве примера метод `hover()`. Вызов `$('a').hover(f,g)` эквивалентен вызову:

```
$('a').bind({mouseenter:f, mouseleave:g});
```

При использовании этой формы метода `bind()` именами свойств передаваемого ему объекта могут быть строки со списками типов событий, разделенных запятыми, включающими пространства имен. Если в этом случае передать второй аргумент, его значение будет использоваться как дополнительные данные для всех связанных обработчиков.

Библиотека jQuery имеет еще один метод регистрации обработчиков. Метод `one()` вызывается и действует подобно методу `bind()`, за исключением того, что зарегистрированный с его помощью обработчик события автоматически удаляется после первого вызова. То есть, как следует из имени метода, обработчики событий, зарегистрированные с помощью `one()`, никогда не вызываются более одного раза.

Единственное, чем не обладают методы `bind()` и `one()`, – это возможность регистрации перехватывающих обработчиков событий, которая поддерживается методом `addEventListener()` (раздел 17.2.3). IE (до версии IE9) не поддерживает перехватывающие обработчики, и библиотека jQuery не пытается имитировать эту особенность.

19.4.5. Удаление обработчиков событий

После регистрации обработчика событий с помощью метода `bind()` (или с помощью простых методов регистрации обработчиков) его можно удалить с помощью метода `unbind()`, чтобы предотвратить его вызов при появлении событий в будущем. (Обратите внимание, что метод `unbind()` удаляет только обработчики, зарегистрированные методом `bind()` и родственными ему методами объекта `jQuery`. Он не удаляет обработчики, зарегистрированные с помощью метода `addEventListener()` или `attachEvent()`, а также не удаляет обработчики, объявленные в атрибутах элементов, таких как `onclick` и `onmouseover`.) При вызове без аргументов метод `unbind()` удалит все обработчики событий (для всех типов событий) из всех выбранных элементов в объекте `jQuery`:

```
$('*').unbind(); // Удалит все обработчики событий из всех элементов!
```

При вызове с единственным строковым аргументом он удалит все обработчики указанного типа события (или типов, если строка содержит несколько имен) из всех выбранных элементов в объекте `jQuery`:

```
// Удалить все обработчики событий mouseover и mouseout во всех элементах <a>
$('a').unbind("mouseover mouseout");
```

Это не самый лучший подход, и его не следует использовать при разработке модулей, потому что пользователь модуля может использовать другие модули, регистрирующие свои собственные обработчики этих же типов событий в тех же элементах. Однако если модуль будет регистрировать обработчики событий с использованием пространств имен, можно будет использовать эту версию вызова метода `unbind()` с одним аргументом для удаления только обработчиков события из вашего пространства или пространств имен:

```
// Удалить все обработчики событий mouseover и mouseout в пространстве имен "myMod"
$('a').unbind("mouseover.myMod mouseout.myMod");
// Удалить только обработчики событий из пространства имен myMod
$('a').unbind(".myMod");
// Удалить обработчик события click из пространств имен "ns1" и "ns2"
$('a').unbind("click.ns1.ns2");
```

Если потребуется удалить только те обработчики событий, которые были зарегистрированы вами, и вы не использовали пространства имен, вы должны получить ссылки на функции обработчиков событий и использовать версию вызова метода `unbind()` с двумя аргументами. В этом случае в первом аргументе передается строка с типом события (без указания пространств имен), а во втором – функция-обработчик:

```
$('#mybutton').unbind('click', myClickHandler);
```

При вызове метода `unbind()` таким способом он удалит указанный обработчик указанного типа (или типов) события из всех выбранных элементов в объекте `jQuery`. Обратите внимание, что обработчики событий могут удаляться версией метода `unbind()` с двумя аргументами, даже когда они были зарегистрированы с дополнительными данными, с помощью версии метода `bind()`, принимающей три аргумента.

Методу `unbind()` можно также передавать объект в единственном аргументе. В этом случае метод `unbind()` будет вызываться рекурсивно для каждого свойства объекта. Имена свойств этого объекта должны определять типы событий, а значения – функции обработчиков:

```
$('a').unbind({ // Удалить конкретные обработчики событий mouseover и mouseout
    mouseover: mouseoverHandler,
    mouseout: mouseoutHandler
});
```

Наконец, существует еще один способ вызова метода `unbind()`. Если передать ему объект `Event`, созданный библиотекой jQuery, он удалит обработчики событий, которым будет передано это событие. Вызов `unbind(ev)` эквивалентен вызову `unbind(ev.type, ev.handler)`.

19.4.6. Возбуждение событий

Зарегистрированные обработчики событий будут вызываться автоматически, когда пользователь начнет использовать мышь или клавиатуру или когда будут возникать события других типов. Однако иногда бывает полезно генерировать события вручную. Проще всего сделать это, вызвав один из простых методов регистрации обработчиков (такой как `click()` или `mouseover()`) без аргументов. Подобно многим методам объекта `jQuery`, которые могут играть роль методов чтения и записи, эти методы регистрируют обработчики событий, когда вызываются с аргументами, и запускают их, когда вызываются без аргументов. Например:

```
$("#my_form").submit(); // Будет действовать, как если бы пользователь щелкнул
                        // на кнопке отправки формы
```

Метод `submit()` в инструкции выше синтезирует объект `Event` и запустит все обработчики событий, которые были зарегистрированы для события «submit». Если ни один из этих обработчиков не вернет `false` или не вызовет метод `preventDefault()` объекта `Event`, форма действительно будет отправлена. Обратите внимание, что события будут всплывать, даже если сгенерировать их вручную, как в данном примере. Это означает, что возбуждение события в множестве выбранных элементов может также привести к вызову обработчиков событий в предках этих элементов.

Важно отметить, что методы объекта `jQuery`, генерирующие события, будут запускать любые обработчики событий, зарегистрированные методами объекта `jQuery`, а также обработчики, объявленные в HTML-атрибутах или свойствах объектов `Element`, таких как `onsubmit`. Но с их помощью невозможно вручную запустить обработчики событий, зарегистрированные методом `addEventListener()` или `attachEvent()` (однако эти обработчики по-прежнему будут вызываться при возникновении настоящих событий).

Отметьте также, что механизм возбуждения событий в библиотеке jQuery является синхронным – в нем не используется очередь событий. Когда событие генерируется вручную, обработчики событий будут вызваны немедленно, до того как метод, возбудивший событие, вернет управление. Если вы генерируете событие «click» и один из запущенных обработчиков сгенерирует событие «submit», все обработчики события «submit» будут вызваны до того, как будет вызван следующий обработчик события «click».

Методы, такие как `submit()`, удобно использовать для связывания и возбуждения событий; но как библиотека jQuery определяет более обобщенный метод `bind()`, так же она определяет и более обобщенный метод `trigger()`. Обычно при вызове в первом аргументе методу `trigger()` передается строка с типом события, и он запускает обработчики, зарегистрированные для этого типа во всех выбранных элементах в объекте `jQuery`. То есть вызов `submit()` в примере выше эквивалентен вызову:

```
$("#my_form").trigger("submit");
```

В отличие от методов `bind()` и `unbind()`, в вызове метода `trigger()` нельзя указать более одного типа события в первом аргументе. Однако, подобно методам `bind()` и `unbind()`, он позволяет указывать пространства имен возбуждаемых событий, чтобы запустить только обработчики, зарегистрированные для этого пространства имен. Если потребуется запустить только обработчики событий, не привязанные ни к какому пространству имен, следует добавить в строку с типом события восклицательный знак. Обработчики, зарегистрированные посредством свойств, таких как `onclick`, считаются обработчиками, не привязанными к какому-либо пространству имен:

```
$("button").trigger("click.ns1"); // Запустит обработчики в пространстве имен
$("button").trigger("click!");    // Запустит обработчики, не привязанные
                                  // к какому-либо пространству имен
```

Вместо строки с типом события в первом аргументе методу `trigger()` можно передать объект `Event` (или любой другой объект, имеющий свойство `type`). Свойство `type` будет определять, какие обработчики должны запускаться. Если передать объект `Event` библиотеки jQuery, этот объект будет передан обработчикам. Если передать простой объект, автоматически будет создан новый объект `Event` библиотеки jQuery и в него будут добавлены свойства объекта, переданного методу. Это самый простой способ передать обработчикам событий дополнительные данные:

```
// Обработчик onclick элемента button1 генерирует то же событие для button2
$('#button1').click(function(e) { $('#button2').trigger(e); });

// Добавит дополнительные свойства в обьект события при возбуждении события
$('#button1').trigger({type:'click', synthetic:true});

// Этот обработчик проверяет дополнительные свойства, чтобы отличить
// настоящее событие от искусственного
$('#button1').click(function(e) { if (e.synthetic) {...}; });
```

Передать дополнительные данные обработчикам при возбуждении событий вручную можно также с помощью второго аргумента метода `trigger()`. Значение, переданное методу `trigger()` во втором аргументе, будет передаваться во втором аргументе всем запущенным обработчикам событий. Если во втором аргументе передать массив, каждый его элемент будет передан обработчикам в виде отдельного аргумента:

```
$('#button1').trigger("click", true);    // Передать единственный дополнительный аргумент
$('#button1').trigger("click", [x,y,z]); // Передать три дополнительных аргумента
```

Иногда может потребоваться запустить все обработчики события данного типа, независимо от того, к какому элементу документа они привязаны. Для этого можно выбрать все элементы вызовом `$('*')` и вызвать метод `trigger()` относительно результата, но это решение весьма неэффективно. Вместо того чтобы возбуждать событие в глобальном масштабе, можно вызвать вспомогательную функцию `jQuery.event.trigger()`. Эта функция принимает те же аргументы, что и метод `trigger()`, и эффективно запускает обработчики событий указанного типа, имеющиеся во всем документе. Обратите внимание, что «глобальные события», возбуждаемые таким способом, не всплывают, и при этом запускаются только обработчики событий, зарегистрированные с использованием методов объекта `jQuery` (обработчики, зарегистрированные с помощью свойств модели DOM, не запускаются).

После вызова обработчиков событий метод `trigger()` (и вспомогательные методы, вызывающие его) выполняет действия, предусмотренные по умолчанию для сгенерированного типа события (предполагается, что ни один обработчик не вернул значение `false` или не вызвал метод `preventDefault()` объекта события). Например, если возбудить событие «submit» в элементе `<form>`, метод `trigger()` вызовет метод `submit()` этой формы, а если возбудить в элементе событие «focus», метод `trigger()` вызовет метод `focus()` этого элемента.

Если необходимо вызвать обработчики событий без выполнения действий по умолчанию, вместо метода `trigger()` следует использовать метод `triggerHandler()`. Этот метод действует точно так же, как и метод `trigger()`, за исключением того, что он сначала вызывает методы `preventDefault()` и `cancelBubble()` объекта `Event`. Это означает, что искусственное событие не будет всплывать, и для него не будут выполняться действия, предусмотренные по умолчанию.

19.4.7. Реализация собственных событий

Система управления событиями в библиотеке jQuery создана на основе стандартных событий, таких как щелчки мышью или нажатия клавиш, генерируемых веб-броузерами. Но она не ограничивается только этими событиями и позволяет использовать любую строку в качестве имени типа события. Метод `bind()` позволяет регистрировать обработчики таких «нестандартных событий», а метод `trigger()` – вызывать эти обработчики.

Такая разновидность косвенного вызова обработчиков нестандартных событий может оказаться весьма полезной при разработке модулей и реализации модели издатель/подписчик или шаблона Observer (наблюдатель). Зачастую при использовании собственных событий может оказаться полезной возможность возбуждать их глобально, с помощью функции `jQuery.event.trigger()` вместо метода `trigger()`:

```
// Когда пользователь щелкнет на кнопке "logoff", отправить собственное событие
// всем подписанным на него наблюдателям, которые должны сохранить информацию
// о своем состоянии, и затем перейти на страницу выхода.
$("#logoff").click(function() {
    $.event.trigger("logoff");     // Отправить широковещательное событие
    window.location = "logoff.php"; // Перейти на другую страницу
});
```

В разделе 19.6.4 вы узнаете, что методы поддержки архитектуры Ajax в библиотеке jQuery аналогичным образом рассылают широковещательные события заинтересованным приемникам.

19.4.8. Динамические события

Метод `bind()` связывает обработчики событий с конкретными элементами документа, подобно методам `addEventListener()` и `attachEvent()` (глава 17). Но веб-приложения, использующие библиотеку jQuery, часто создают новые элементы динамически. Если мы воспользуемся методом `bind()` для привязки обработчика событий ко всем элементам `<a>`, имеющимся в документе, и затем создадим новые элементы `<a>`, эти новые элементы не будут иметь обработчиков событий, которые были в старых элементах, и будут вести себя иначе.

В библиотеке jQuery эта проблема решается с помощью «динамических событий». Чтобы задействовать динамические события, вместо методов `bind()` и `unbind()` следует использовать методы `delegate()` и `undelegate()`.

Обычно метод `delegate()` вызывается относительно `$(document)`, и ему передаются строка селектора, строка с типом события и функция обработчика, а он регистрирует внутренний обработчик в объекте документа или окна (или в любом другом элементе, находящемся в объекте `jQuery`). Когда событие указанного типа всплывет до этого внутреннего обработчика, он выяснит, соответствует ли целевой элемент события (элемент, в котором оно возникло) строке селектора, и вызовет указанную функцию обработчика. То есть, чтобы обеспечить обработку события «mouseover» и в старых, и во вновь созданных элементах `<a>`, можно зарегистрировать обработчик, как показано ниже:

```
$(document).delegate("a", "mouseover", linkHandler);
```

Или сначала применить метод `bind()` к статической части документа, а затем с помощью метода `delegate()` обработать динамически изменяемую часть:

```
// Статические обработчики событий для статических ссылок
$("a").bind("mouseover", linkHandler);

// Динамические обработчики событий для фрагментов документа,
// которые изменяются динамически
$(".dynamic").delegate("a", "mouseover", linkHandler);
```

Подобно тому, как метод `bind()` имеет версию с тремя аргументами, позволяющую указать значение свойства `data` объекта события, метод `delegate()` имеет версию с четырьмя аргументами, позволяющую то же самое. При использовании этой версии дополнительные данные следует передавать в третьем аргументе, а функцию обработчика – в четвертом.

Важно понимать, что динамические события основаны на механизме всплытия. К тому моменту, когда оно всплывет до объекта документа, оно может пройти через множество статических обработчиков. А если какой-либо из этих обработчиков вызовет метод `cancelBubble()` объекта `Event`, динамический обработчик так и не будет вызван.

Объект `jQuery` имеет метод `live()`, который также можно использовать для регистрации динамических обработчиков событий. Метод `live()` устроен немного сложнее, чем метод `delegate()`, но он, как и метод `bind()`, имеет версии с двумя и тремя аргументами, которые чаще всего используются на практике. Два вызова метода `delegate()`, показанные выше, можно было бы заменить следующими вызовами метода `live()`:

```
$("a").live("mouseover", linkHandler);
$("a", $(".dynamic")).live("mouseover", linkHandler);
```

Когда вызывается метод `live()`, элементы, находящиеся в объекте `jQuery`, в действительности никак не используются. Что имеет значение, так это строка селектора и объект контекста (первый и второй аргументы функции `$()`), использовавшиеся при создании объекта `jQuery`. Эти значения доступны в виде свойств `selector` и `context` объектов `jQuery` (раздел 19.1.2). Обычно функция `$()` вызывается с единственным аргументом, а роль контекста в этом случае играет текущий документ. То есть при использовании объекта `x` типа `jQuery` следующие две строки можно считать эквивалентными:

```
x.live(type,handler);
$(x.context).delegate(x.selector, type, handler);
```

Для удаления динамических обработчиков событий используются методы die() и undelegate(). Метод die() может вызываться с одним или с двумя аргументами. Если методу передать единственный аргумент, определяющий тип события, он удалит все динамические обработчики событий, соответствующие селектору и типу событий. А если передать тип события и функцию обработчика, он удалит только указанный обработчик. Например:

```
$('a').die('mouseover');              // Удалит все динамические обработчики
                                      // события mouseover из элементов <a>
$('a').die('mouseover', linkHandler); // Удалит только указанный динамический обработчик
```

Метод undelegate() действует аналогично методу die(), но более явно отделяет контекст (элементы, в которых был зарегистрирован внутренний обработчик) и строку селектора. Вызовы метода die() выше можно было заменить вызовами метода undelegate(), как показано ниже:

```
$(document).undelegate('a');    // Удалит все динамические обработчики из элементов <a>
$(document).undelegate('a', 'mouseover'); // Удалит динамические обработчики
                                          // события mouseover
$(document).undelegate('a', 'mouseover', linkHandler); // Указанный обработчик
```

Наконец, метод undelegate() может также вызываться вообще без аргументов. В этом случае он удаляет все динамические обработчики, привязанные к выбранным элементам.

19.5. Анимационные эффекты

В главе 16 демонстрировалось, как можно управлять стилями CSS в элементах документа. Например, устанавливая CSS-свойство visibility, можно заставлять элементы появляться и скрываться. В разделе 16.3.1 было показано, как, управляя стилями CSS, можно воспроизводить анимационные эффекты. Например, вместо того, чтобы просто сделать элемент невидимым, можно постепенно уменьшать значение его свойства opacity в течение половины секунды и заставить его исчезать плавно. Подобные визуальные эффекты оставляют у пользователей более приятные впечатления, и библиотека jQuery упрощает их реализацию.

Объект jQuery определяет методы воспроизведения основных визуальных эффектов, такие как fadeIn() и fadeOut(). Кроме них он определяет также метод animate(), позволяющий воспроизводить более сложные эффекты. В следующих подразделах описываются и методы воспроизведения простых эффектов, и более сложный универсальный метод animate(). Однако для начала познакомимся с некоторыми общими особенностями механизма анимационных эффектов в библиотеке jQuery.

Каждый анимационный эффект имеет продолжительность, которая определяет, как долго должен продолжаться эффект. Продолжительность можно указать в виде числа миллисекунд или в виде строки. Строка «fast» означает 200 миллисекунд. Строка «slow» означает 600 миллисекунд. Если указать строку, которая не будет распознана библиотекой jQuery, по умолчанию будет использована продолжительность 400 миллисекунд. Имеется возможность определять новые названия, обозначающие продолжительность, добавляя новые отображения строк в числа в объект jQuery.fx.speeds:

```
jQuery.fx.speeds["medium-fast"] = 300;
jQuery.fx.speeds["medium-slow"] = 500;
```

Методы воспроизведения эффектов объекта `jQuery` обычно принимают продолжительность в первом необязательном аргументе. Если опустить этот аргумент, по умолчанию продолжительность будет составлять 400 миллисекунд. Однако некоторые методы, не получив аргумент с продолжительностью, выполняют операцию немедленно, без анимационного эффекта:

```
$("#message").fadeIn();        // Эффект проявления будет длиться 400 мсек
$("#message").fadeOut("fast"); // Эффект растворения будет длиться 200 мсек
```

Эффекты в библиотеке jQuery воспроизводятся асинхронно. Когда производится вызов метода анимационного эффекта, такого как `fadeIn()`, он сразу же возвращает управление, а воспроизведение эффекта выполняется «в фоновом режиме». Поскольку методы анимационных эффектов возвращают управление до того, как эффект завершится, многие из них принимают во втором аргументе (также необязательном) функцию, которая будет вызвана по окончании воспроизведения эффекта. Этой функции не передается никаких аргументов, но ссылка `this` в ней будет указывать на элемент документа, к которому применялся эффект. Функция будет вызвана по одному разу для каждого выбранного элемента:

```
// Быстро проявить элемент, а когда он станет видимым, вывести в нем текст.
$("#message").fadeIn("fast", function() { $(this).text("Привет, Мир!"); });
```

Передача функции обратного вызова методу воспроизведения эффекта позволяет выполнять действия по его окончании. Однако в этом нет необходимости, когда требуется просто последовательно воспроизвести несколько эффектов. По умолчанию библиотека jQuery ставит анимационные эффекты в очередь (в разделе 19.5.2.2 демонстрируется, как изменить это поведение по умолчанию). Если вызвать метод анимационного эффекта относительно элемента, для которого уже воспроизводится анимационный эффект, воспроизведение нового эффекта не начнется немедленно, а будет отложено до окончания воспроизведения текущего эффекта. Например, можно заставить элемент «моргать», пока он не проявится окончательно:

```
$("#blinker").fadeIn(100).fadeOut(100).fadeIn(100).fadeOut(100).fadeIn();
```

Методы анимационных эффектов объекта `jQuery` принимают необязательные аргументы, определяющие продолжительность и функцию обратного вызова. Однако этим методам можно также передавать объект, свойства которого определяют параметры эффектов:

```
// Передать продолжительность и функцию не в отдельных аргументах, а в свойствах объекта
$("#message").fadeIn({
    duration: "fast",
    complete: function() { $(this).text("Привет, Мир!"); }
});
```

Этот прием с передачей объекта обычно применяется при использовании универсального метода `animate()`, но он также может применяться и при работе с методами простых анимационных эффектов. Использование объекта позволяет также определять и другие, расширенные параметры, такие как параметры управления очередью и переходами эффектов. Доступные параметры будут описаны в разделе 19.5.2.2.

Отключение анимационных эффектов

Анимационные эффекты стали нормой на многих веб-сайтах, но они нравятся не всем пользователям: некоторые считают их раздражающими, а кто-то даже может испытывать неприятные ощущения. Пользователи с ограниченными возможностями могут обнаружить, что анимационные эффекты затрудняют использование вспомогательных технологий, таких как программы чтения с экрана, а владельцы устаревших компьютеров будут ощущать нехватку вычислительных мощностей. В качестве жеста уважения к своим пользователям вы должны стараться использовать более простые анимационные эффекты и в небольшом количестве, а также предоставлять возможность полностью отключать их. Библиотека jQuery дает простую возможность отключить сразу все эффекты: достаточно просто установить свойство `jQuery.fx.off` в значение `true`. В результате продолжительность всех эффектов будет установлена равной 0 миллисекунд, что заставит их выполняться мгновенно, без анимации.

Чтобы дать пользователям возможность отключать анимационные эффекты, можно включить в сценарий следующий фрагмент:

```
$(".stopmoving").click(function() { jQuery.fx.off = true; });
```

Затем веб-дизайнер должен включить в страницу элемент с классом «stopmoving», щелчок на котором будет отключать воспроизведение анимационных эффектов.

19.5.1. Простые эффекты

Библиотека jQuery реализует девять методов простых анимационных эффектов скрытия и отображения элементов. Их можно разделить на три группы по типам воспроизводимых ими эффектов:

`fadeIn()`, `fadeOut()`, `fadeTo()`

Это самые простые эффекты: методы `fadeIn()` и `fadeOut()` просто управляют CSS-свойством `opacity`, чтобы скрыть элемент или сделать его видимым. Оба принимают необязательные аргументы, определяющие продолжительность и функцию обратного вызова. Метод `fadeTo()` несколько отличается: он принимает аргумент, определяющий конечное значение непрозрачности и плавно изменяет текущее значение непрозрачности элемента до указанного. В первом обязательном аргументе методу `fadeTo()` передается продолжительность (или объект с параметрами), а во втором обязательном аргументе – конечное значение непрозрачности. Функция обратного вызова передается в третьем необязательном аргументе.

`show()`, `hide()`, `toggle()`

Метод `fadeOut()`, описанный выше, делает элемент невидимым, но сохраняет занимаемую им область в документе. Метод `hide()`, напротив, удаляет элемент из потока документа, как если бы его CSS-свойство `display` было установлено в значение `none`. При вызове без аргументов методы `hide()` и `show()` просто немедленно скрывают и отображают выбранные элементы. Однако при вызове с аргументом, определяющим продолжительность (или объект с параметрами), они воспроизводят анимационный эффект скрытия или появления. Метод `hide()` уменьшает ширину и высоту элемента до 0 и одновременно уменьшает до 0 непрозрачность элемента. Метод `show()` выполняет обратные действия.

Метод `toggle()` изменяет состояние видимости элементов: для скрытых элементов он вызывает метод `show()`, а для видимых – метод `hide()`. Как и при работе с методами `show()` и `hide()`, чтобы воспроизвести анимационный эффект, методу `toggle()` необходимо передать продолжительность или объект с параметрами. Передача значения `true` методу `toggle()` эквивалентна вызову метода `show()` без аргументов, а передача значения `false` – вызову метода `hide()` без аргументов. Обратите также внимание, что если передать методу `toggle()` одну или более функций, он зарегистрирует обработчики событий, как описывалось в разделе 19.4.1.

`slideDown()`, `slideUp()`, `slideToggle()`

Метод `slideUp()` скрывает выбранные элементы в объекте `jQuery`, постепенно уменьшая их высоту до 0, и затем устанавливает CSS-свойство `display` в значение «none». Метод `slideDown()` выполняет противоположные действия, чтобы сделать скрытый элемент видимым. Метод `slideToggle()` переключает состояние видимости элементов, используя методы `slideUp()` и `slideDown()`. Каждый из этих трех методов принимает необязательные аргументы, определяющие продолжительность и функцию обратного вызова (или объект с параметрами).

Следующий пример демонстрирует использование методов из всех трех групп. Имейте в виду, что по умолчанию библиотека jQuery ставит анимационные эффекты в очередь, что обеспечивает их выполнение по очереди:

```
// Растворить все элементы, затем показать их, затем свернуть и развернуть
$("img").fadeOut().show(300).slideUp().slideToggle();
```

Различные расширения библиотеки jQuery (раздел 19.9) добавляют в нее дополнительные анимационные эффекты. Наиболее полный набор эффектов включает библиотека jQuery UI (раздел 19.10).

19.5.2. Реализация собственных анимационных эффектов

Метод `animate()` позволяет воспроизводить более сложные анимационные эффекты, чем методы простых эффектов. Первый аргумент метода `animate()` определяет воспроизводимый эффект, а остальные аргументы – параметры этого эффекта. Первый аргумент является обязательным: это должен быть объект, свойства которого задают CSS-атрибуты и их конечные значения. Метод `animate()` плавно изменяет CSS-свойства всех элементов от текущих их значений до указанного конечного значения. То есть эффект, воспроизводимый описанным выше методом `slideUp()`, можно, например, реализовать, как показано ниже:

```
// Уменьшить высоту всех изображений до 0
$("img").animate({ height: 0 });
```

Во втором необязательном аргументе методу `animate()` можно передать объект с параметрами эффекта:

```
$("#sprite").animate({
   opacity: .25,                 // Изменить непрозрачность до 0,25
   font-size: 10                 // Изменить размер шрифта до 10 пикселов
}, {
   duration: 500,                // Продолжительность 1/2 секунды
   complete: function() {        // Вызвать эту функцию по окончании
      this.text("До свидания");  // Изменить текст в элементе.
   }
});
```

Вместо объекта с параметрами во втором аргументе метод `animate()` позволяет также передать три наиболее часто используемых параметра в виде отдельных аргументов. Во втором аргументе можно передать продолжительность (в виде числа или строки), в третьем аргументе – имя функции, выполняющей переходы (подробнее об этой функции рассказывается чуть ниже.) И в четвертом аргументе – функцию обратного вызова.

В самом общем случае метод `animate()` принимает два аргумента с объектами. Первый определяет CSS-атрибуты, которые будут изменяться, а второй – параметры их изменения. Чтобы полностью понять, как выполняются анимационные эффекты в библиотеке jQuery, необходимо поближе познакомиться с обоими объектами.

19.5.2.1. Объект, определяющий изменяемые атрибуты

Первым аргументом метода `animate()` должен быть объект. Имена свойств этого объекта должны совпадать с именами CSS-атрибутов, а значения этих свойств должны определять конечные значения атрибутов, которые должны быть получены к окончанию эффекта. Участвовать в анимационном эффекте могут только атрибуты с числовыми значениями: невозможно реализовать плавное изменение значения цвета, шрифта или свойств-перечислений, таких как `display`. Если значением свойства является число, подразумевается, что оно измеряется в пикселах. Если значение является строкой, в ней можно указать единицы измерения. Если единицы измерения отсутствуют, опять же предполагается, что значение измеряется в пикселах. Чтобы указать относительные значения, в строковые значения следует добавить префикс: «+=» – для увеличения и «-=» – для уменьшения значения. Например:

```
$("p").animate({
   "margin-left": "+=.5in", // Увеличить отступ абзаца
   opacity: "-=.1"          // Уменьшить непрозрачность
});
```

Обратите внимание на кавычки, окружающие имя свойства «margin-left» в примере литерала объекта выше. Наличие дефиса в имени этого свойства делает его недопустимым идентификатором в языке JavaScript, поэтому в подобных случаях следует использовать кавычки. Разумеется, библиотека jQuery позволяет также использовать альтернативные имена со смешанным регистром символов, такие как `marginLeft`.

Помимо числовых значений (с необязательными единицами измерения и префиксами «+=» и «-=») существует еще три значения, которые можно использовать в объектах, определяющих изменяемые свойства. Значение «hide» сохранит текущее значение указанного свойства и затем плавно изменит его до 0. Значение «show» плавно изменит значение CSS-свойства до его сохраненного значения. При использовании значения «show» библиотека jQuery вызовет метод show() по завершении эффекта. А при использовании значения «hide» она вызовет метод hide().

Можно также использовать значение «toggle», которое обеспечит увеличение («show») или уменьшение («hide») значения атрибута в зависимости от его текущего состояния. Например, ниже показано, как можно реализовать эффект «slideRight» сворачивания вправо (подобный эффекту сворачивания вверх, воспроизводимому методом slideUp(), но изменяющий ширину элемента):

```
$("img").animate({
    width: "hide",
    borderLeft: "hide",
    borderRight: "hide",
    paddingLeft: "hide",
    paddingRight: "hide"
});
```

Замените значения свойств на «show» или «toggle», чтобы получить эффект разворачивания по горизонтали, аналогичные тем, что воспроизводятся методами slideDown() и slideToggle().

19.5.2.2. Объект с параметрами анимационного эффекта

Во втором необязательном аргументе методу animate() может передаваться объект с параметрами анимационного эффекта. Вы уже знакомы с двумя наиболее важными параметрами. Значением свойства duration может быть число, определяющее длительность эффекта в миллисекундах, а также строка «fast», «slow» или любая другая, объявленная в свойстве jQuery.fx.speeds.

Другим параметром, с которым вы уже встречались, является свойство complete: оно определяет функцию, которая должна быть вызвана по окончании эффекта. Похожее свойство step определяет функцию, которая должна вызываться для каждого шага или кадра анимации. Элемент, к которому применяется эффект, передается этой функции в виде значения ссылки this, а текущее значение изменяемого свойства – в первом аргументе.

Свойство queue объекта с параметрами определяет – должен ли данный эффект ставиться в очередь. То есть должно ли откладываться воспроизведение данного эффекта до окончания всех предыдущих эффектов. По умолчанию все анимационные эффекты ставятся в очередь. Если свойству queue присвоить значение false, эффект не будет поставлен в очередь. Воспроизведение таких внеочередных эффектов начинается немедленно. Последующие анимационные эффекты, которые ставятся в очередь, не будут ждать завершения внеочередных эффектов. Рассмотрим следующий пример:

```
$("img").fadeIn(500)
        .animate({"width":"+=100"}, {queue:false, duration:1000})
        .fadeOut(500);
```

Эффекты, запускаемые методами fadeIn() и fadeOut(), будут поставлены в очередь, а эффект, запускаемый вызовом метода animate() (эффект изменения значения свойства width на протяжении 1000 миллисекунд) – нет. Изменение ширины начнется одновременно с эффектом fadeIn(). Эффект fadeOut() начнется сразу после окончания эффекта fadeIn(): он не будет ждать, пока завершится эффект, изменяющий ширину элемента.

Функции переходов

В самом простом случае воспроизведение анимационного эффекта заключается в линейном изменении во времени значения свойства. Например, через 100 миллисекунд после начала эффекта, длительность которого составляет 400 миллисекунд, величина изменения значения свойства составит 25%. То есть при линейном изменении свойства `opacity` от 1,0 до 0,0 (как, например, при использовании метода `fadeOut()`) в этот момент оно должно иметь значение 0,75. Однако, как оказывается, визуальные эффекты дают более глубокие впечатления, если они выполняются нелинейно. Поэтому библиотека jQuery предусматривает возможность использования «функции перехода», которая отображает проценты от общего времени выполнения эффекта в проценты от конечного значения свойства. Библиотека jQuery передает функции перехода значение времени в диапазоне от 0 до 1, а она должна вернуть другое значение в диапазоне от 0 до 1, исходя из которого библиотека jQuery вычислит значение CSS-свойства, опираясь на его вычисленное значение. Конечно, в общем случае ожидается, что функции переходов будут возвращать значение 0, когда им передается значение 0, и 1, когда им передается значение 1, но между этими двумя значениями они могут быть нелинейными, что будет проявляться в ускорении и замедлении анимационных эффектов.

По умолчанию в библиотеке jQuery используется синусоидальная функция перехода: эффект сначала протекает медленно, затем ускоряется, и затем опять замедляется при приближении к конечному значению. Функции переходов в библиотеке jQuery имеют имена. Функция по умолчанию называется «swing», а линейная функция называется «linear». Вы можете добавлять свои функции переходов в объект `jQuery.easing`:

```
jQuery.easing["squareroot"] = Math.sqrt;
```

Библиотека jQuery UI и расширение, известное как «the jQuery Easing Plugin», определяют весьма исчерпывающий набор дополнительных функций переходов.

Остальные параметры анимационных эффектов имеют отношение к функциям переходов. Свойство `easing` объекта с параметрами определяет имя функции перехода. По умолчанию библиотека jQuery использует синусоидальную функцию с именем «swing». Если необходимо, чтобы анимационный эффект воспроизводился линейно, следует использовать параметры, как показано ниже:

```
$("img").animate({"width":"+=100"}, {duration: 500, easing:"linear"});
```

Напомню, что параметры `duration`, `easing` и `complete` можно также передавать методу `animate()` в виде отдельных аргументов. То есть предыдущий анимационный эффект можно запустить так:

```
$("img").animate({"width":"+=100"}, 500, "linear");
```

Наконец, механизм воспроизведения анимационных эффектов в библиотеке jQuery позволяет даже указывать для разных CSS-свойств разные функции переходов. Сделать это можно двумя разными способами, как показано в следующем примере:

```
// Требуется скрыть изображения, подобно методу hide(), при этом изменение
// размеров изображения должно протекать линейно, а изменение непрозрачности -
// с применением функции перехода "swing" по умолчанию

// Первый способ:
// Использовать параметр specialEasing, чтобы указать другую функцию перехода
$("img").animate({ width:"hide", height:"hide", opacity:"hide" },
                 { specialEasing: { width: "linear", height: "linear" }});

// Второй способ:
// Передать массивы [целевое значение, функция перехода] в объекте,
// который передается в первом аргументе.
$("img").animate({
    width: ["hide", "linear"], height: ["hide", "linear"], opacity:"hide"
});
```

19.5.3. Отмена, задержка и постановка эффектов в очередь

В библиотеке jQuery определяется еще несколько методов, имеющих отношение к анимационным эффектам и очередям, которые необходимо знать. Первым из них является метод `stop()`: он останавливает воспроизведение текущего анимационного эффекта для выбранных элементов. Метод `stop()` принимает два необязательных логических аргумента. Если в первом аргументе передать `true`, очередь анимационных эффектов для выбранных элементов будет очищена: т. е. вместе с текущим эффектом будут отменены все остальные эффекты, находящиеся в очереди. По умолчанию этот аргумент принимает значение `false`: если аргумент не указан, эффекты, находящиеся в очереди, не отменяются. Второй аргумент определяет, должны ли изменяемые CSS-свойства остаться в текущем состоянии или им должны быть присвоены конечные значения. Значение `true` во втором аргументе заставляет присвоить им конечные значения. Значение `false` (или отсутствие аргумента) оставляет текущие значения CSS-свойств.

Когда анимационные эффекты запускаются по событиям, возникающим в результате действий пользователя, может потребоваться отменить все текущие и запланированные анимационные эффекты, прежде чем запустить новые. Например:

```
// Сделать изображения непрозрачными, когда указатель мыши находится над ними.
// Не забудьте отменить все запланированные анимационные эффекты по событиям мыши!
$("img").bind({
    mouseover: function() { $(this).stop().fadeTo(300, 1.0); },
    mouseout: function() { $(this).stop().fadeTo(300, 0.5); }
});
```

Второй метод, связанный с анимацией, который мы рассмотрим здесь, – это метод `delay()`. Он просто добавляет задержку в очередь эффектов. В первом аргументе он принимает задержку в миллисекундах (или строку), а во втором необязательном аргументе – имя очереди (указывать второй аргумент обычно не требуется: об именах очередей будет рассказываться ниже). Метод `delay()` можно использовать в составных анимационных эффектах, как показано ниже:

```
// Быстро растворить элемент до половины, подождать, а затем свернуть его
$("img").fadeTo(100, 0.5).delay(200).slideUp();
```

В примере выше, где применялся метод `stop()`, были использованы обработчики событий «mouseover» и «mouseout» для плавного изменения непрозрачности изображений. Этот пример можно усовершенствовать, если добавить в него короткую задержку перед запуском анимационного эффекта. При таком подходе, если указатель мыши быстро пересекает изображение без остановки, никакого анимационного эффекта не возникает:

```
$("img").bind({
    mouseover: function() { $(this).stop(true).delay(100).fadeTo(300, 1.0); },
    mouseout: function() { $(this).stop(true).fadeTo(300, 0.5); }
});
```

Последняя группа методов, с которыми мы познакомимся, предоставляют низкоуровневый доступ к механизму очередей в библиотеке jQuery. Очереди в библиотеке jQuery реализованы в виде списков функций, которые выполняются в порядке их следования. Каждая очередь связана с определенным элементом документа (или с объектом `Document` или `Window`), и все очереди в элементах не зависят друг от друга. Добавить новую функцию в очередь можно с помощью метода `queue()`. Когда функция достигнет головы очереди, она будет автоматически исключена из очереди и вызвана. Внутри этой функции ключевое слово `this` ссылается на элемент, с которым связана данная очередь. В единственном аргументе функции будет передана другая функция. Когда функция из очереди завершит свою работу, она должна вызвать переданную ей функцию; тем самым она запустит следующую операцию из очереди. Если эта функция не будет вызвана, обработка очереди будет заморожена и оставшиеся в ней функции никогда не будут вызваны.

Мы уже знаем, что методам анимационных эффектов можно передавать функции обратного вызова для выполнения некоторых действий по завершении воспроизведения эффекта. Того же эффекта можно добиться, поместив эту функцию в очередь:

```
// Проявить элемент, подождать, записать в него текст, и изменить его рамку
$("#message").fadeIn().delay(200).queue(function(next) {
    $(this).text("Привет, Мир!");      // Вывести текст
    next();                            // Запустить следующую функцию в очереди
}).animate({borderWidth: "+=10px;"}); // Увеличить толщину рамки
```

Аргумент с функцией, который передается функции, помещаемой в очередь, – это новая особенность, появившаяся в версии jQuery 1.4. При использовании более ранних версий библиотеки функции в очереди извлекали следующую функцию «вручную», вызывая метод `dequeue()`:

```
$(this).dequeue(); // Вместо next()
```

Если очередь пуста, метод `dequeue()` ничего не делает. В противном случае он удаляет функцию, находящуюся в голове очереди, и вызывает ее, устанавливая значение ссылки `this` и передавая функцию, описанную выше.

Кроме того, существует еще несколько методов управления очередью вручную. Метод `clearQueue()` очищает очередь. Если вместо единственной функции передать методу `queue()` массив функций, он заменит текущую очередь новым массивом функций. А вызов метода `queue()` без аргумента вернет текущую очередь в виде массива. Кроме того, библиотека jQuery определяет версии методов `queue()` и `dequeue()` в виде вспомогательных функций. Например, если потребуется добавить функцию `f` в очередь элемента `e`, сделать это можно будет с помощью метода или функции:

```
$(e).queue(f);      // Создать объект jQuery, хранящий e, и вызвать метод queue
jQuery.queue(e,f); // Просто вызвать вспомогательную функцию jQuery.queue()
```

Наконец, обратите внимание, что методы `queue()`, `dequeue()` и `clearQueue()` принимают необязательный первый аргумент с именем очереди. Методы анимационных эффектов используют очередь с именем «fx» и именно эта очередь используется по умолчанию, если имя очереди не указано явно. Механизм очередей в библиотеке jQuery с успехом можно использовать для выполнения асинхронных операций в определенной последовательности: вместо того чтобы передавать функцию обратного вызова каждой асинхронной операции, чтобы она запускала следующую функцию в последовательности, для управления последовательностью можно использовать очередь. Просто используйте имя очереди, отличное от «fx», и не забывайте, что функции в очереди не вызываются автоматически. Чтобы запустить первую функцию в очереди, необходимо явно вызвать метод `dequeue()`, а по завершении каждая операция должна запускать следующую.

19.6. Реализация Ajax в библиотеке jQuery

Ajax – популярное название комплекса приемов разработки веб-приложений, в которых применяются возможности использования протокола HTTP (глава 18) для загрузки данных по мере необходимости без перезагрузки страниц. Приемы Ajax оказались настолько полезны в современных веб-приложениях, что в библиотеку jQuery были включены вспомогательные функции, реализующие и упрощающие их использование. Библиотека jQuery определяет один высокоуровневый вспомогательный метод и четыре высокоуровневые вспомогательные функции. Все они основаны на одной низкоуровневой функции, `jQuery.ajax()`. В следующих подразделах мы сначала познакомимся с высокоуровневыми утилитами, а затем детально рассмотрим функцию `jQuery.ajax()`. Понимание принципов действия функции `jQuery.ajax()` совершенно необходимо, чтобы до конца понять, как действуют высокоуровневые утилиты, даже если вам никогда не придется использовать ее явно.

19.6.1. Метод load()

Метод `load()` является самой простой из всех утилит в библиотеке jQuery: он принимает URL-адрес, асинхронно загружает его содержимое и затем вставляет это содержимое в каждый из выбранных элементов, заменяя любое имеющееся содержимое. Например:

```
// Загружать и отображать последнюю информацию о состоянии каждые 60 сек.
setInterval(function() { $("#stats").load("status_report.html"); }, 60000);
```

Коды состояния Ajax, генерируемые библиотекой jQuery

Все утилиты поддержки архитектуры Ajax в библиотеке jQuery, включая метод `load()`, вызывают функции обратного вызова, чтобы асинхронно уведомить приложение об успехе или неудаче. Во втором аргументе этим функциям обратного вызова передается строка с одним из следующих значений:

«success»

Указывает, что запрос выполнен успешно.

«notmodified»

Эта строка говорит о том, что запрос выполнен нормально, но сервер вернул HTTP-ответ 304 «Not Modified», сообщая, что запрошенный документ не изменился с момента предыдущего запроса. Этот код состояния возвращается, только когда параметр `ifModified` имеет значение `true` (подробнее об этом в разделе 19.6.3.1.). Версия jQuery 1.4 интерпретирует код «notmodified» как успех, но в более ранних версиях он интерпретировался как ошибка.

«error»

Указывает, что запрос завершился неудачей из-за возникновения какой-либо HTTP-ошибки. Чтобы получить более подробную информацию об ошибке, можно проверить код состояния HTTP в объекте `XMLHttpRequest`, который также передается функции.

«timeout»

Если Ajax-запрос не завершился в течение указанного вами интервала времени, функции обратного вызова будет передан этот код состояния. По умолчанию Ajax-запросы в библиотеке jQuery не имеют ограничений по времени выполнения. Этот код состояния может быть получен, только если был установлен параметр `timeout` (раздел 19.6.3.1).

«parsererror»

Этот код состояния говорит о том, что HTTP-запрос завершился успешно, но библиотека jQuery не смогла разобрать ответ. Этот код возвращается, например, когда сервер отправляет XML-документ или JSON-текст, сформированный с ошибками. Обратите внимание, что строка имеет значение «parsererror», а не «parseerror».

Мы уже встречались с методом `load()` в разделе 19.4.1, где использовали его для регистрации обработчика события «load». Если в первом аргументе этому методу передать функцию, а не строку, он будет играть роль метода регистрации обработчика события, а не метода поддержки архитектуры Ajax.

Если вам потребуется просто отобразить фрагмент загруженного документа, добавьте в конец URL-адреса пробел и селектор jQuery. Когда содержимое URL-адреса будет загружено, указанный селектор будет использован для выбора фрагмента загруженного HTML-документа и его отображения:

```
// Загрузить и отобразить температуру из документа со сводкой погоды
$('#temp').load("weather_report.html #temperature");
```

Обратите внимание, что селектор в конце этого URL-адреса похож на идентификатор фрагмента (части URL-адресов, начинающиеся с символа решетки, описывались в разделе 14.2). Пробел в данном случае является обязательной частью, если необходимо, чтобы библиотека jQuery вставила лишь выбранный фрагмент (или фрагменты) загруженного документа.

В дополнение к обязательному URL-адресу метод `load()` принимает два необязательных аргумента. Первый – данные, добавляемые в URL-адрес или отправляемые вместе с запросом. Если в этом аргументе передать строку, она будет добавлена в конец URL (при необходимости после `?` или `&`). Если передать объект, он будет преобразован в строку пар имя/значение, разделенных амперсандами, и отправлен вместе с запросом. (Описание особенностей преобразования объекта в строку для использования в составе запроса Ajax приводится во врезке в разделе 19.6.2.2). Обычно метод `load()` выполняет HTTP-запрос методом GET, но, если передать объект с данными, будет выполнен запрос методом POST. Например:

```
// Загрузить сводку погоды для населенного пункта с указанным почтовым индексом
$('#temp').load("us_weather_report.html", "zipcode=02134");

// Здесь данные передаются в виде объекта, в котором дополнительно указывается,
// что температура должна возвращаться в градусах по шкале Фаренгейта
$('#temp').load("us_weather_report.html", { zipcode:02134, units:'F' });
```

В другом необязательном аргументе методу `load()` можно передать функцию, которая будет вызвана в случае завершения Ajax-запроса, успешного или нет, и (в случае успеха) после того, как содержимое URL-адреса будет загружено и вставлено в выбранные элементы. Если методу не передаются никакие дополнительные данные, эту функцию обратного вызова можно передать во втором аргументе. Иначе она должна передаваться в третьем аргументе. Указанная функция будет вызвана как метод для каждого элемента, находящегося в объекте jQuery, и ей будет передано три аргумента: полное содержимое, загруженное с указанного URL-адреса, строка с кодом состояния и объект `XMLHttpRequest`, использовавшийся для загрузки содержимого. Аргумент с кодом состояния – это код состояния, созданный библиотекой jQuery, а не возвращаемый протоколом HTTP, и он может быть строкой, такой как «success», «error» или «timeout».

19.6.2. Вспомогательные функции поддержки Ajax

Другие высокоуровневые утилиты поддержки архитектуры Ajax в библиотеке jQuery являются функциями, а не методами и вызываются относительно глобального имени `jQuery` или `$`, а не относительно объекта с выбранными элементами. Функция `jQuery.getScript()` загружает и выполняет файлы со сценариями на языке JavaScript. Функция `jQuery.getJSON()` загружает содержимое URL и разбирает его как текст в формате JSON, а получившийся в результате объект передает указанной функции обратного вызова. Обе эти функции вызывают функцию `jQuery.get()`, которая является более универсальной функцией загрузки данных из указанного URL-адреса. Наконец, функция `jQuery.post()` действует подобно функции `jQuery.get()`, но выполняет HTTP-запрос методом POST, а не GET. Как и метод `load()`, все эти функции выполняются асинхронно: они возвращают управление

еще до того, как будут загружены какие-либо данные, и извещают программу о результатах посредством указанной функции обратного вызова.

19.6.2.1. jQuery.getScript()

Функция `jQuery.getScript()` принимает в первом аргументе URL-адрес файла со сценарием на языке JavaScript. Она асинхронно загружает и выполняет этот сценарий в глобальной области видимости. Выполняться могут сценарии как общего происхождения с документом, так и сторонние:

```
// Динамически загрузить сценарий с некоторого другого сервера
jQuery.getScript("http://example.com/js/widget.js");
```

Во втором аргументе можно передать функцию обратного вызова, и в этом случае библиотека jQuery вызовет ее сразу после того, как загруженный сценарий будет выполнен.

```
// Загрузить библиотеку и воспользоваться ею после загрузки
jQuery.getScript("js/jquery.my_plugin.js", function() {
    $('div').my_plugin(); // Воспользоваться загруженной библиотекой
});
```

Для получения текста сценария, который должен быть выполнен, функция `jQuery.getScript()` обычно использует объект `XMLHttpRequest`. Но для выполнения междоменных запросов (когда сценарий поставляется сервером, отличным от того, откуда был получен текущий документ), библиотека jQuery использует элемент `<script>` (раздел 18.2). Если запрос удовлетворяет ограничениям политики общего происхождения, в первом аргументе функции обратного вызова передается текст сценария, во втором – код состояния «success» и в третьем – объект `XMLHttpRequest`, использовавшийся для получения текста сценария. Возвращаемым значением функции `jQuery.getScript()` в данном случае также является объект `XMLHttpRequest`. Для междоменных запросов, которые выполняются без участия объекта `XMLHttpRequest`, текст сценария не сохраняется. В этом случае в первом и третьем аргументах функции обратного вызова передается значение `undefined`, и возвращаемым значением функции `jQuery.getScript()` также является значение `undefined`.

Функция обратного вызова, передаваемая функции `jQuery.getScript()`, вызывается только в случае успешного выполнения запроса. Если также необходимо получить извещение в случае ошибки, следует использовать низкоуровневую функцию `jQuery.ajax()`. То же относится и к трем другим вспомогательным функциям, описываемым в этом разделе.

19.6.2.2. jQuery.getJSON()

Функция `jQuery.getJSON()` подобна функции `jQuery.getScript()`: она загружает текст и затем обрабатывает его особым образом перед вызовом указанной функции обратного вызова. Функция `jQuery.getJSON()` не выполняет загруженный текст как сценарий, а выполняет синтаксический разбор этого текста как данных в формате JSON (используя функцию `jQuery.parseJSON()`: описывается в разделе 19.7). Функцию `jQuery.getJSON()` имеет смысл использовать, только когда ей передается функция обратного вызова. Если содержимое URL было благополучно загружено и разобрано, как данные в формате JSON, то полученный в результате объект передается функции обратного вызова в первом аргументе. Как и при использова-

нии функции `jQuery.getScript()`**, во втором и третьем аргументах передаются код состояния «success» и объект** `XMLHttpRequest`**:**

```
// Допустим, что data.json содержит текст: '{"x":1,"y":2}'
jQuery.getJSON("data.json", function(data) {
    // Здесь data - это объект {x:1, y:2}
});
```

Передача данных утилитам поддержки Ajax в библиотеке jQuery

Большинство методов поддержки архитектуры Ajax в библиотеке jQuery принимают аргумент (или параметр), определяющий данные для отправки на сервер вместе с URL. Обычно эти данные принимают вид строки, закодированной в формате URL, пар имя/значение, отделяющихся друг от друга символами амперсанда. (Этот формат представления данных известен, как MIME-тип «application/x-www-form-urlencoded». Его можно рассматривать как аналог формата JSON – формата представления простых JavaScript-объектов в виде строк.) При выполнении HTTP-запросов методом GET эта строка с данными добавляется в конец URL-адреса запроса. При выполнении запросов методом POST она отправляется в теле запроса после отправки HTTP-заголовков.

Получить строку с данными в этом формате можно с помощью метода `serialize()` **объекта** `jQuery`**, содержащего формы или элементы формы. Отправить, например, HTML-форму с помощью метода** `load()` **можно следующим образом:**

```
$("#submit_button").click(function(event) {
    $(this.form).load(            // Заменить форму, загрузив...
        this.form.action,         // из указанного url
        $(this.form).serialize()); // с данными, добавленными в него
    event.preventDefault();       // Отменить отправку формы по умолч.
    this.disabled = "disabled";   // Предотвратить несколько
});                               // попыток отправки
```

Если в аргументе (или параметре) передать функции поддержки архитектуры Ajax в библиотеке jQuery объект, а не строку, то библиотека jQuery по умолчанию (с исключениями, описываемыми ниже) автоматически преобразует объект в строку, вызвав функцию `jQuery.param()`**. Эта вспомогательная функция интерпретирует свойства объекта как пары имя/значение и, например, преобразует объект** `{x:1,y:"hello"}` **в строку** `"x=1&y=hello"`**.**

В версии jQuery 1.4 функция `jQuery.param()` **способна обрабатывать более сложные объекты. Если значение свойства объекта является массивом, для каждого элемента этого массива будет создана отдельная пара имя/значение, а к имени свойства будут добавлены квадратные скобки. Если значением свойства является объект, имена свойств этого вложенного объекта помещаются в квадратные скобки и добавляются к имени внешнего свойства. Например:**

```
$.param({a:[1,2,3]})        // Вернет "a[]=1&a[]=2&a[]=3"
$.param({o:{x:1,y:true}})  // Вернет "o[x]=1&o[y]=true"
$.param({o:{x:{y:[1,2]}}}) // Вернет "o[x][y][]=1&o[x][y][]=2"
```

Для обратной совместимости с версией jQuery 1.3 и ниже во втором аргументе функции `jQuery.param()` можно передать значение `true` или установить параметр `traditional` в значение `true`. Это предотвратит использование расширенных возможностей сериализации свойств, значениями которых являются массивы или объекты.

Иногда бывает необходимо передать в теле POST-запроса объект `Document` (или какой-то другой объект, который не должен преобразовываться автоматически). В этом случае можно установить в параметре `contentType` тип данных и в параметре `processData` значение `false` и тем самым предотвратить передачу объекта с данными функции `jQuery.param()`.

В отличие от `jQuery.getScript()`, функция `jQuery.getJSON()` принимает необязательный аргумент с данными, подобный тому, что передается методу `load()`. Если функции `jQuery.getJSON()` необходимо передать данные, они должны передаваться во втором аргументе, а функция обратного вызова – в третьем. Если дополнительные данные не требуются, функцию обратного вызова можно передать во втором аргументе. Если данные являются строкой, она будет добавлена в конец URL-адреса, вслед за символом `?` или `&`. Если данные передаются в виде объекта, он будет преобразован в строку (как описывается во врезке) и добавлен в конец URL-адреса.

Если строка URL или данных, передаваемая функции `jQuery.getJSON()`, содержит последовательность символов «=?» в конце или перед амперсандом, она определяет запрос JSONP. (Описание формата JSONP приводится в разделе 18.2.) Библиотека jQuery заменит знак вопроса именем функции обратного вызова, которая будет создана автоматически, и функция `jQuery.getJSON()` будет действовать, как если бы выполнялся запрос сценария, а не объекта в формате JSON. Этот прием не работает со статическими JSON-файлами данных: он может применяться только при наличии сценариев на стороне сервера, поддерживающих формат JSONP. Однако, поскольку данные в формате JSONP обрабатываются как сценарии, для их получения допускается выполнять междоменные запросы.

19.6.2.3. jQuery.get() и jQuery.post()

Функции `jQuery.get()` и `jQuery.post()` загружают содержимое из указанного адреса URL, отправляя дополнительные данные, если они имеются, и передавая результат указанной функции обратного вызова. Функция `jQuery.get()` делает это, выполняя HTTP-запрос методом GET, а функция `jQuery.post()` – методом POST, но во всем остальном эти две вспомогательные функции совершенно идентичны. Обе они принимают те же три аргумента, что и функция `jQuery.getJSON()`: обязательный URL-адрес, необязательную строку или объект с данными и технически необязательную, но практически всегда используемую функцию обратного вызова. В первом аргументе функции обратного вызова передаются полученные данные, во втором – строка «success» и в третьем – объект `XMLHttpRequest` (если он использовался для выполнения запроса):

```
// Запросить текст с сервера и отобразить его в диалоге alert
jQuery.get("debug.txt", alert);
```

Помимо трех аргументов, описанных выше, эти две функции могут принимать четвертый необязательный аргумент (передается как третий аргумент, если дополнительные данные отсутствуют), который определяет тип запрашиваемых данных. Этот аргумент влияет на обработку полученных данных перед передачей их функции обратного вызова. Метод `load()` использует тип «html», `jQuery.getScript()` – тип «script», а `jQuery.getJSON()` – тип «json». Однако функции `jQuery.get()` и `jQuery.post()` более гибкие, чем эти специализированные утилиты, и им можно указать любой из этих типов. Допустимые значения этого аргумента и особенности поведения библиотеки jQuery при его отсутствии описываются во врезке.

Типы данных, поддерживаемые реализацией Ajax в библиотеке jQuery

Функциям `jQuery.get()` и `jQuery.post()` допускается передавать любой из шести типов данных. Кроме того, как описывается ниже, любой из этих типов можно также передавать функции `jQuery.ajax()` в виде параметра `dataType`:

`"text"`

Вернуть ответ сервера как простой текст, без дополнительной обработки.

`"html"`

Этот тип обрабатывается так же, как тип `"text"`: ответ возвращается как простой текст. Этот тип используется методом `load()`, который вставляет текст ответа в документ.

`"xml"`

Предполагает, что URL-адрес ссылается на данные в формате XML, и для их получения вместо свойства `responseText` объекта `XMLHttpRequest` библиотека jQuery использует свойство `responseXML`. Функции обратного вызова передается не строка с текстом документа, а объект `Document`, представляющий XML-документ.

`"script"`

Предполагает, что URL-адрес ссылается на файл со сценарием на языке JavaScript, и перед передачей функции обратного вызова текст ответа выполняется как сценарий. Этот тип используется функцией `jQuery.getScript()`. Когда указывается тип `"script"`, библиотека будет выполнять междоменные запросы с помощью элемента `<script>` вместо объекта `XMLHttpRequest`.

`"json"`

Предполагает, что URL-адрес ссылается на файл с данными в формате JSON. Функции обратного вызова в этом случае передается объект, полученный в результате разбора содержимого ответа с помощью функции `jQuery.parseJSON()` (раздел 19.7). Этот тип используется функцией `jQuery.getJSON()`. Если указан тип `"json"` и строка URL или данных содержит `"=?"`, тип преобразуется в `"jsonp"`.

`"jsonp"`

Предполагает, что URL-адрес ссылается на серверный сценарий, поддерживающий протокол JSONP передачи данных в формате JSON в виде аргумента указанной функции на стороне клиента. (Подробнее о формате JSONP рассказывается в разделе 18.2.) Когда указывается этот тип, функции обратного вызова передается разобранный объект. Поскольку JSONP-запросы могут выполняться с помощью элементов `<script>`, этот тип можно использовать для выполнения междоменных запросов, подобно типу `"script"`. При использовании этого типа строка URL или данных обычно должна включать параметр вида `"&jsonp=?"` или `"&callback=?"`. Библиотека jQuery заменит знак вопроса именем автоматически созданной функции обратного вызова. (Обратите внимание на параметры `jsonp` и `jsonpCallback`, описываемые в разделе 19.6.3.3, позволяющие определить альтернативные варианты.)

Если при вызове `jQuery.get()`, `jQuery.post()` или `jQuery.ajax()` не указан ни один из этих типов, библиотека jQuery проверит заголовок «Content-Type» HTTP-ответа. Если этот заголовок включает подстроку «xml», функции обратного вызова будет передан XML-документ. Иначе, если заголовок включает подстроку «json», ответ будет разобран, как данные в формате JSON, и полученный объект будет передан функции обратного вызова. Иначе, если заголовок включает подстроку «JavaScript», ответ будет выполнен как сценарий. Иначе данные будут интерпретироваться как простой текст.

19.6.3. Функция jQuery.ajax()

Все утилиты поддержки архитектуры Ajax в библиотеке jQuery в конечном итоге вызывают `jQuery.ajax()` – самую сложную функцию во всей библиотеке. Функция `jQuery.ajax()` принимает всего один аргумент: объект с параметрами, свойства которого определяют детали, касающиеся выполнения Ajax-запроса. Вызов `jQuery.getScript(url,callback)`, например, эквивалентен следующему вызову функции `jQuery.ajax()`:

```
jQuery.ajax({
    type: "GET",            // Метод HTTP-запроса.
    url: url,               // URL-адрес запрашиваемых данных.
    data: null,             // Не добавлять дополнительные данные в URL.
    dataType: "script",     // Выполнить ответ как сценарий.
    success: callback       // Вызвать эту функцию по завершении.
});
```

Эти пять фундаментальных параметров можно также установить при использовании функций `jQuery.get()` и `jQuery.post()`. Однако при непосредственном использовании `jQuery.ajax()` имеется возможность указать большое количество других параметров. Все параметры (включая пять основных, представленных выше) детально описываются ниже.

Прежде чем погрузиться в описание параметров, обратите внимание, что имеется возможность определить значения по умолчанию любых из этих параметров, передав объект с параметрами функции `jQuery.ajaxSetup()`:

```
jQuery.ajaxSetup({
    timeout: 2000, // Прерывать все Ajax-запросы через 2 секунды
    cache: false   // Игнорировать кэш броузера, добавляя время в URL
});
```

После выполнения программного кода, приведенного выше, указанные параметры `timeout` и `cache` будут действовать для всех Ajax-запросов (включая высокоуровневые утилиты, такие как `jQuery.get()` и `load()`), при вызове которых не указываются значения этих параметров.

В процессе знакомства с многочисленными параметрами и особенностями функций обратного вызова в следующих разделах может оказаться полезным еще раз ознакомиться с информацией во врезках, касающейся кодов состояния Ajax-запросов и типов данных в разделах 19.6.1 и 19.6.2.3.

Поддержка Ajax в версии jQuery 1.5

В версии jQuery 1.5, которая вышла, когда эта книга готовилась к печати, модуль поддержки архитектуры Ajax был полностью переписан, и в нем появилось несколько новых удобных особенностей. Самое важное, что функция `jQuery.ajax()` и все утилиты поддержки Ajax, описанные выше, теперь возвращают объект `jqXHR`. Этот объект имитирует прикладной интерфейс объекта `XMLHttpRequest` даже для запросов (например, выполняемых функцией `$.getScript()`), не использующих объект `XMLHttpRequest`. Кроме того, объект `jqXHR` определяет методы `success()` и `error()`, которые можно использовать для регистрации функций, вызываемых в случае успешного или неудачного завершения запроса. То есть вместо того чтобы передавать функцию обратного вызова функции `jQuery.get()`, например, ее можно зарегистрировать с помощью метода `success()` объекта `jqXHR`, возвращаемого этой утилитой:

```
jQuery.get("data.txt")
    .success(function(data) { console.log("Получено ", data); })
    .success(function(data) { process(data); });
```

19.6.3.1. Часто используемые параметры

Ниже перечислены параметры, которые наиболее часто передаются функции `jQuery.ajax()`:

`type`

Определяет метод HTTP-запроса. По умолчанию имеет значение «GET». Другим наиболее часто используемым значением является «POST». Допускается указывать также другие методы HTTP-запросов, такие как «DELETE» и «PUT», но они поддерживаются не всеми броузерами. Обратите внимание, что имя этого параметра может вводить в заблуждение: он не имеет никакого отношения к типу данных в запросе или ответе, и для него лучше подошло бы имя «method».

`url`

URL-адрес загружаемых данных. При выполнении GET-запросов параметр `data` добавляется в конец этого URL-адреса. Библиотека jQuery может автоматически добавлять параметры в строку URL при выполнении JSONP-запросов и когда параметр `cache` имеет значение `false`.

`data`

Данные, добавляемые в конец URL-адреса (для GET-запросов) или отправляемые в теле запроса (для POST-запросов). Может быть строкой или объектом. Объекты обычно преобразуются в строки, как описывалось во врезке в разделе 19.6.2.2, однако имеются некоторые исключения, которые приводятся в описании параметра `processData`.

`dataType`

Определяет тип ожидаемых данных в ответе и способ их обработки библиотекой jQuery. Допустимыми значениями являются «text», «html», «script», «json», «jsonp» и «xml». Суть этих значений описана во врезке в разделе 19.6.2.3. Этот параметр не имеет значения по умолчанию. Если он не указан, библиотека jQuery проверит заголовок «Content-Type» ответа, чтобы определить, что делать с полученными данными.

`contentType`

Определяет HTTP-заголовок «Content-Type» запроса. По умолчанию имеет значение «application/x-www-form-urlencoded», которое обычно используется HTML-формами и большинством серверных сценариев. Если вы установили параметр `type` в значение «POST» и собираетесь отправить в теле запроса простой текст или XML-документ, вам должны также установить этот параметр.

`timeout`

Предельное время ожидания в миллисекундах. Если этот параметр установлен и запрос не завершится в течение указанного времени, выполнение запроса прервется и будет вызвана функция обратного вызова `error` с кодом состояния «timeout». По умолчанию параметр `timeout` имеет значение 0, которое означает, что выполнение запроса будет продолжаться до его завершения и никогда не будет прервано.

`cache`

При выполнении GET-запросов, если этот параметр будет установлен в значение `false`, библиотека jQuery добавит параметр `_=` в строку URL или заменит существующий параметр с этим именем. В качестве значения этого параметра будет установлено текущее время (в виде количества миллисекунд). Это предотвратит использование кэша броузера, поскольку каждый раз, когда запрос будет выполняться, строка URL будет иной.

`ifModified`

Когда этот параметр имеет значение `true`, библиотека jQuery будет сохранять значения заголовков «Last-Modified» и «If-None-Match» ответов для каждого запрошенного URL-адреса и затем будет устанавливать эти заголовки во всех последующих запросах к тем же самым URL-адресам. Это предписывает серверу отправлять HTTP-ответ 304 «Not Modified», если содержимое по указанному URL-адресу не изменилось с момента последнего обращения. По умолчанию данный параметр не установлен и библиотека jQuery не сохраняет эти заголовки.

Библиотека jQuery преобразует HTTP-ответ 304 в код состояния «notmodified». Код «notmodified» не считается ошибкой и передается функции обратного вызова `success` вместо обычного кода состояния «success». То есть если вы устанавливаете параметр `ifModified`, вы также должны проверять код состояния в своей функции обратного вызова – если будет получен код состояния «notmodified», то первый аргумент функции (данные из ответа) будет иметь значение `undefined`. Обратите внимание, что в версии jQuery ниже 1.4 HTTP-ответ 304 интерпретировался как ошибка, и код состояния «notmodified» передавался функции обратного вызова `error`, а не `success`. Подробнее о кодах состояния рассказывается во врезке в разделе 19.6.1.

`global`

Этот параметр определяет, должна ли библиотека jQuery возбуждать события в ходе выполнения Ajax-запроса. По умолчанию имеет значение `true`. Присвойте этому параметру значение `false`, чтобы запретить все события, связанные с поддержкой архитектуры Ajax. (Полное описание событий приводится в разделе 19.6.4.) Имя этого параметра несколько обескураживает: он имеет имя «global», потому что обычно библиотека jQuery возбуждает события глобально, а не в конкретном объекте.

19.6.3.2. Функции обратного вызова

Следующие параметры определяют функции, вызываемые на разных стадиях в ходе выполнения Ajax-запроса. С параметром `success` вы уже знакомы: это функция обратного вызова, которая передается методам, таким как `jQuery.getJSON()`. Обратите внимание, что библиотека jQuery также посылает извещения в ходе выполнения Ajax-запроса в виде событий (если параметр `global` не был установлен в значение `false`).

`context`

Этот параметр определяет объект, используемый в качестве контекста – значения ссылки `this` – для различных функций обратного вызова. Данный параметр не имеет значения по умолчанию, и если его не устанавливать, функции обратного вызова будут вызываться в контексте объекта с параметрами, в котором они определяются. Значение параметра `context` также воздействует на порядок возбуждения событий механизмом поддержки Ajax (раздел 19.6.4). Значением этого параметра должен быть объект `Window`, `Document` или `Element`, в котором могут возбуждаться события.

`beforeSend`

Этот параметр определяет функцию, которая должна вызываться перед отправкой Ajax-запроса на сервер. Первым аргументом этой функции передается объект `XMLHttpRequest`, а вторым – объект с параметрами запроса. Функция `beforeSend` дает программе возможность установить собственные HTTP-заголовки в объекте `XMLHttpRequest`. Если эта функция вернет `false`, выполнение Ajax-запроса будет прервано. Обратите внимание, что для выполнения междуменных запросов типов «script» и «jsonp» объект `XMLHttpRequest` не используется и функция, определяемая параметром `beforeSend`, не вызывается.

`success`

Этот параметр определяет функцию, которая должна вызываться в случае успешного выполнения Ajax-запроса. В первом аргументе ей передаются данные, отправленные сервером. Во втором аргументе – код состояния, сгенерированный библиотекой jQuery, и в третьем – объект `XMLHttpRequest`, использовавшийся для выполнения запроса. Как описывалось в разделе 19.6.2.3, тип данных в первом аргументе зависит от значения параметра `dataType` или заголовка «Content-Type» в ответе сервера. Если данные имеют тип «xml», в первом аргументе передается объект `Document`. Если данные имеют тип «json» или «jsonp», в первом аргументе передается объект, полученный в результате разбора ответа сервера в формате JSON. Если данные имеют тип «script», ответом является текст загруженного сценария (однако к моменту вызова функции сценарий уже будет выполнен, поэтому в данном случае ответ обычно игнорируется). Для других типов ответ интерпретируется как простой текст, содержащийся в запрошенном ресурсе.

Код состояния во втором аргументе обычно является строкой «success». Но, если был установлен параметр `ifModified`, в этом аргументе может также передаваться строка «notmodified». В этом случае сервер не отправляет данные в ответе, и в первом аргументе будет передано значение `undefined`. Для выполнения междоменных запросов на получение данных типов «script» и «jsonp» используется элемент `<script>`, а не объект `XMLHttpRequest`, поэтому для таких запросов в третьем аргументе вместо объекта `XMLHttpRequest` будет передаваться значение `undefined`.

`error`

Этот параметр определяет функцию, которая должна вызываться в случае неудачи Ajax-запроса. В первом аргументе этой функции передается объект `XMLHttpRequest` запроса (если таковой использовался). Во втором аргументе – код состояния, сгенерированный библиотекой jQuery. Это может быть строка «error» – в случае ошибки протокола HTTP, «timeout» – в случае превышения времени ожидания и «parsererror» – в случае ошибки, возникшей в ходе разбора ответа сервера. Например, если XML-документ или объект в формате JSON будет сформирован неправильно, функция получит код состояния «parsererror». В этом случае в третьем аргументе функции `error` будет передан объект `Error`, представляющий исключение. Обратите внимание, что запросы с параметром `dataType="script"`, возвращающие недопустимый программный код JavaScript, не вызывают ошибки. Любые ошибки в сценарии просто игнорируются и вместо функции `error` вызывается функция `success`.

`complete`

Этот параметр определяет функцию, которая должна вызываться по завершении Ajax-запроса. Каждый Ajax-запрос завершается либо успехом и вызывает функцию `success`, либо неудачей и вызывает функцию `error`. Библиотека jQuery вызывает функцию `complete` после вызова функции `success` или `error`. В первом аргументе функции `complete` передается объект `XMLHttpRequest`, а во втором – код состояния.

19.6.3.3. Редко используемые параметры и обработчики

Следующие параметры используются довольно редко. Некоторые из них являются параметрами, которые вам едва ли придется устанавливать, а другие определяют обработчики для тех, кому потребуется изменить порядок обработки Ajax-запросов, используемый в библиотеке jQuery по умолчанию.

`async`

Запросы HTTP по своей природе являются асинхронными. Однако объект `XMLHttpRequest` дает возможность заблокировать выполнение сценария до получения ответа. Если присвоить этому параметру значение `false`, библиотека jQuery будет блокировать работу сценария. Данный параметр не влияет на значение, возвращаемое функцией `jQuery.ajax()`: она всегда возвращает объект `XMLHttpRequest`, если он используется. При выполнении синхронных запросов вы можете самостоятельно извлекать ответ сервера и код состояния HTTP из объекта `XMLHttpRequest` или определить функцию обратного вызова `complete` (как в случае асинхронных запросов), чтобы получить разобранный ответ и код состояния jQuery.

`dataFilter`

Этот параметр определяет функцию фильтрации или предварительной обработки данных, возвращаемых сервером. В первом аргументе ей будут передаваться необработанные данные, полученные от сервера (либо в виде строки, либо в виде объекта `Document`, при запросе XML-документа), а во втором аргументе – значение параметра `dataType`. Эта функция должна возвращать значение, которое будет использоваться вместо ответа сервера. Обратите внимание, что функция `dataFilter` вызывается перед разбором данных в формате JSON или перед выполнением сценария. Кроме того, отметьте, что `dataFilter` не вызывается при выполнении междоменных запросов данных типов «script» и «jsonp».

`jsonp`

Когда параметр `dataType` имеет значение «jsonp», значение параметра `url` или `data` обычно включает параметр строки запроса вида «jsonp=?». Если библиотека jQuery не обнаружит этот параметр в URL-адресе или в данных, она вставит его, используя значение параметра `jsonp` в качестве имени параметра в строке запроса. По умолчанию параметр `jsonp` имеет значение «callback». Присвойте ему другое значение, если сервер, поддерживающий обмен данными в формате JSONP, ожидает получить другое имя параметра в строке запроса и вы явно не указываете это имя в строке URL или в данных. Подробнее о формате JSONP рассказывается в разделе 18.2.

`jsonpCallback`

Для запросов с параметром `dataType`, имеющим значение «jsonp» (или «json», когда URL-адрес включает параметр строки запроса, такой как «jsonp=?»), библиотека jQuery будет изменять строку URL, подставляя вместо знака вопроса имя функции-обертки, которой сервер будет передавать данные. Обычно библиотека jQuery синтезирует уникальное имя функции, опираясь на текущее время. Присвойте этому параметру свое значение, если вам потребуется явно указать собственную функцию. Но имейте в виду, что в этом случае библиотека jQuery не будет вызывать функции обратного вызова `success` и `complete` и не будет возбуждать обычные события.

`processData`

Когда значением параметра `data` является объект (или объект передается во втором аргументе функции `jQuery.get()` и родственным ей функциям), библиотека jQuery обычно преобразует этот объект в строку в формате «application/x-www-form-urlencoded» (как описывается во врезке в разделе 19.6.2.2). Если потребуется предотвратить это преобразование (например, чтобы передать объект `Document` в теле POST-запроса), присвойте этому параметру значение `false`.

`scriptCharset`

Для междоменных запросов данных типов «script» и «jsonp», при выполнения которых используется элемент `<script>`, этот параметр определяет значение атрибута `charset` элемента. Он никак не влияет на обычные запросы, выполняемые с помощью объекта `XMLHttpRequest`.

`traditional`

В библиотеке jQuery версии 1.4 несколько изменился способ сериализации объектов с данными в строки формата «application/x-www-form-urlencoded» (подробности приводятся во врезке в разделе 19.6.2.2). Присвойте этому параметру значение `true`, если необходимо, чтобы библиотека jQuery использовала прежний порядок.

`username`, `password`

Если для выполнения запроса необходимо выполнить процедуру аутентификации пользователя, укажите имя пользователя и пароль в этих двух параметрах.

`xhr`

Этот параметр определяет фабричную функцию, создающую объект `XMLHttpRequest`. Она вызывается без аргументов и должна возвращать объект, реализующий прикладной программный интерфейс объекта `XMLHttpRequest`. Этот весьма низкоуровневый обработчик позволяет создавать собственные обертки вокруг объекта `XMLHttpRequest` и добавлять новые особенности или расширять его методы.

19.6.4. События в архитектуре Ajax

В разделе 19.6.3.2 говорилось, что функция `jQuery.ajax()` имеет четыре параметра, определяющие функции обратного вызова: `beforeSend`, `success`, `error` и `complete`. Помимо вызова этих функций, функции поддержки архитектуры Ajax в библиотеке jQuery также возбуждают собственные события на каждой стадии выполнения запроса. В следующей таблице перечислены параметры, определяющие функции обратного вызова, и соответствующие им события:

Функция обратного вызова	Тип события	Метод регистрации обработчика
`beforeSend`	«ajaxSend»	`ajaxSend()`
`success`	«ajaxSuccess»	`ajaxSuccess()`
`error`	«ajaxError»	`ajaxError()`
`complete`	«ajaxComplete»	`ajaxComplete()`
	«ajaxStart»	`ajaxStart()`
	«ajaxStop»	`ajaxStop()`

Зарегистрировать обработчики этих событий можно с помощью метода `bind()` (раздел 19.4.4), используя строку с типом события из второй колонки, или с помощью методов из третьей колонки. Метод `ajaxSuccess()` и другие действуют точно так же, как `click()`, `mouseover()` и другие простые методы регистрации событий, о которых рассказывалось в разделе 19.4.1.

Поскольку события, генерируемые реализацией архитектуры Ajax, являются нестандартными и генерируются самой библиотекой jQuery, а не броузером, объект `Event`, передаваемый обработчикам, не содержит сколько-нибудь полезной информации. Однако вместе со всеми событиями – «ajaxSend», «ajaxSuccess», «ajaxError» и «ajaxComplete» – передаются дополнительные аргументы. Всем обработчикам этих событий будет передаваться два дополнительных аргумента. В первом дополнительном аргументе будет передаваться объект `XMLHttpRequest`, а во втором – объект с параметрами. Это означает, что обработчик события, например «ajaxSend», сможет добавлять собственные заголовки в объект `XMLHttpRequest`, подобно функции обратного вызова `beforeSend`. Обработчикам события «ajaxError» передается третий дополнительный аргумент помимо двух, только что описанных. В этом последнем аргументе будет передаваться объект `Error`, если таковой имеется, который был создан в результате возникшей ошибки. Довольно странно, но обработчикам событий архитектуры Ajax не передается код состояния, генерируемый библиотекой jQuery. Если, например, в обработчике события «ajaxSuccess» потребуется отличать состояния «success» и «notmodified», необходимо будет проверить код состояния HTTP-ответа в объекте `XMLHttpRequest`.

Последние два события, перечисленные в таблице выше, отличаются от других тем, что не имеют соответствующих им функций обратного вызова, а также тем, что их обработчикам не передаются дополнительные аргументы. «ajaxStart» и «ajaxStop» – это пара событий, которые извещают о начале и окончании выполнения сетевых операций при выполнении Ajax-запроса. Когда библиотека jQuery не выполняет ни одного Ajax-запроса и инициируется новый запрос, она возбуждает событие «ajaxStart». Если до того, как завершится первый запрос, будут запущены новые запросы, эти новые запросы не будут вызывать появление нового события «ajaxStart». Событие «ajaxStop» генерируется, когда завершится последний Ajax-запрос и при этом библиотека jQuery уже не выполняет никаких сетевых операций. Эта пара событий может пригодиться для отображения и сокрытия анимированного сообщения «Загрузка...» или изображения, свидетельствующего о выполнении сетевых операций. Например:

```
$("#loading_animation").bind({
    ajaxStart: function() { $(this).show(); },
    ajaxStop: function() { $(this).hide(); }
});
```

Обработчики событий «ajaxStart» и «ajaxStop» можно связать с любым элементом документа: библиотека jQuery генерирует их глобально (раздел 19.4.6), а не для какого-то конкретного элемента. Другие четыре события архитектуры Ajax – «ajaxSend», «ajaxSuccess», «ajaxError» и «ajaxComplete» – также обычно генерируются глобально, поэтому их обработчики также можно связать с любым элементом документа. Однако если установить параметр `context` при вызове функции `jQuery.ajax()`, эти четыре события будут генерироваться только в контексте указанного элемента.

Наконец, запомните, что появление всех событий архитектуры Ajax в библиотеке jQuery можно предотвратить, присвоив параметру `global` значение `false`. Несмотря на обескураживающее имя параметра `global`, присваивание ему значения `false` предотвращает возбуждение событий не только в глобальном масштабе, но и в объекте `context`.

19.7. Вспомогательные функции

Библиотека jQuery определяет множество вспомогательных функций (и два свойства), которые могут вам пригодиться в ваших программах. Как вы увидите в списке ниже, для многих из этих функций теперь имеются эквиваленты в стандарте ECMAScript 5 (ES5). Функции в библиотеке jQuery были созданы еще до появления стандарта ES5 и действуют во всех броузерах. Ниже в алфавитном порядке перечислены вспомогательные функции:

`jQuery.browser`

Свойство `browser` является не функцией, а объектом, который можно использовать для определения типа броузера (раздел 13.4.5). Если сценарий выполняется в IE, свойство `msie` этого объекта будет иметь значение `true`. В Firefox и родственных ему броузерах значение `true` будет иметь свойство `mozilla`. В Safari и Chrome значение `true` будет иметь свойство `webkit`, а в броузере Opera значение `true` будет иметь свойство `opera`. В дополнение к этим свойствам объект `browser` имеет также свойство `version`, содержащее номер версии броузера. Приема определения типа броузера лучше стараться избегать, насколько это возможно, тем не менее это свойство можно использовать обхода ошибок, характерных для разных броузеров, как показано ниже:

```
if ($.browser.mozilla && parseInt($.browser.version) < 4) {
    // Здесь обрабатывается гипотетическая ошибка в Firefox...
}
```

`jQuery.contains()`

Эта функция принимает в аргументах два элемента документа. Она возвращает `true`, если первый элемент содержит второй, иначе возвращает значение `false`.

`jQuery.each()`

В отличие от метода `each()`, который выполняет итерации только по объектам `jQuery`, вспомогательная функция `jQuery.each()` способна выполнять итерации по элементам массива или свойствам объекта. В первом аргументе она принимает массив или объект, по которому выполняются итерации. Во втором аргументе принимается функция, которая должна быть вызвана для каждого элемента массива или свойства объекта. Этой функции передаются два аргумента: индекс элемента массива или имя свойства объекта и значение элемента массива или свойства объекта. Значение ссылки `this` в этой функции совпадает со значением второго аргумента. Если функция вернет `false`, `jQuery.each()` сразу же вернет управление, не завершив итерации. Функция `jQuery.each()` всегда возвращает значение первого аргумента.

Перечисление свойств функцией `jQuery.each()` выполняется в том же порядке, что и в обычном цикле `for/in`, т. е. в итерациях участвуют все перечислимые свойства, включая унаследованные. Перечисление элементов массива функцией `jQuery.each()` выполняется в порядке следования их индексов, и она не пропускает неопределенные элементы в разреженных массивах.

`jQuery.extend()`

Эта функция принимает объекты в своих аргументах. Она копирует свойства из второго и всех последующих объектов в первый объект, затирая все одноименные свойства в первом объекте. Она пропускает все свойства, имеющие значение `undefined` или `null`. Если передать ей только один объект, свойства этого объекта будут скопированы в сам объект `jQuery`. Возвращаемым значением функции является объект, в который копировались свойства. Если в первом аргументе передать значение `true`, будет выполнено глубокое, или рекурсивное, копирование: второй аргумент будет дополнен свойствами третьего (и всех последующих) объектов.

Эту функцию удобно использовать для создания копий объектов и объединения объектов с параметрами с настройками по умолчанию:

```
var clone = jQuery.extend({}, original);
var options = jQuery.extend({}, default_options, user_options);
```

`jQuery.globalEval()`

Эта функция выполняет строку с программным кодом на языке JavaScript в глобальном контексте, как если бы она была содержимым элемента `<script>`. (В действительности эта функция создает элемент `<script>` и временно вставляет его в документ.)

`jQuery.grep()`

Эта функция похожа на метод `filter()` объекта `Array`, определяемый стандартом ES5. В первом аргументе она принимает массив, а во втором – функцию-предикат и вызывает эту функцию для каждого элемента массива, передавая ей значение и индекс элемента. Функция `jQuery.grep()` возвращает новый массив, содержащий только те элементы аргумента-массива, для которых функция-предикат вернула значение `true` (или другое значение, которое оценивается как истинное). Если в третьем аргументе передать функции `jQuery.grep()` значение `true`, она будет инвертировать возвращаемое значение функции-предиката и вернет массив элементов, для которых функция-предикат вернула ложное значение.

`jQuery.inArray()`

Эта функция похожа на метод `indexOf()` объекта `Array`, определяемый стандартом ES5. В первом аргументе она принимает произвольное значение, а во втором – массив (или объект, подобный массиву) и возвращает индекс первого элемента в массиве, имеющего это значение, или -1, если указанное значение отсутствует в массиве.

`jQuery.isArray()`

Возвращает `true`, если аргумент является объектом `Array`.

`jQuery.isEmptyObject()`

Возвращает `true`, если аргумент не содержит перечислимых свойств.

`jQuery.isFunction()`

Возвращает `true`, если аргумент является объектом `Function`. Обратите внимание, что в IE версии 8 и ниже такие методы, определяемые броузером, как `Window.alert()` и `Element.attachEvent()`, не являются функциями в этом смысле.

`jQuery.isPlainObject()`

Возвращает `true`, если аргумент является «простым» объектом, а не экземпляром некоторого более специализированного типа или класса объектов.

`jQuery.makeArray()`

Если аргумент является объектом, подобным массиву, эта функция скопирует элементы из этого объекта в новый (истинный) массив и вернет этот массив. Если аргумент не является объектом, подобным массиву, эта функция просто вернет новый массив с аргументом в качестве единственного элемента.

`jQuery.map()`

Эта функция похожа на метод `map()` объекта `Array`, определяемый стандартом ES5. В первом аргументе она принимает массив или объект, подобный массиву, а во втором – функцию. Она передает указанной функции значение и индекс каждого элемента массива и возвращает новый массив, содержащий значения, возвращаемые функцией. `jQuery.map()` имеет пару отличий от метода `map()` в стандарте ES5. Если ваша функция отображения вернет `null`, это значение не будет включено в массив с результатами. И если ваша функция отображения вернет массив, в результат будут добавлены элементы этого массива по отдельности, а не сам массив.

`jQuery.merge()`

Эта функция принимает два массива или объекта, подобных массивам, добавляет элементы второго массива в первый и возвращает первый массив. Первый массив изменяется, а второй – нет. Обратите внимание, что эту функцию можно использовать для поверхностного копирования массивов, как показано ниже:

```
var clone = jQuery.merge([], original);
```

`jQuery.parseJSON()`

Эта функция разбирает строку в формате JSON и возвращает результат. Она возбуждает исключение, если в исходной строке будет обнаружена ошибка. Библиотека jQuery использует стандартную версию функции `JSON.parse()`, если она определена в броузере. Обратите внимание, что в библиотеке jQuery имеется только функция разбора строк в формате JSON, но в ней отсутствует функция сериализации объектов в формат JSON.

`jQuery.proxy()`

Эта функция напоминает метод `bind()` (раздел 8.7.4) объекта `Function`, определяемый стандартом ES5. В первом аргументе она принимает функцию, а во втором – объект и возвращает новую функцию, которая вызывает оригинальную как метод указанного объекта. Она не выполняет частичное применение аргументов, как метод `bind()`.

Функция `jQuery.proxy()` может также вызываться с объектом в первом аргументе и именем свойства во втором. Значение свойства с указанным именем должно быть функцией. В этом случае вызов функции `jQuery.proxy(o,n)` вернет то же, что и вызов `jQuery.proxy(o[n],o)`.

Функция `jQuery.proxy()` предназначена для использования с механизмом связывания обработчиков событий в библиотеке jQuery. Если в качестве обработчика была связана функция, полученная с помощью `jQuery.proxy()`, то удалить ее можно, указав ее оригинал.

`jQuery.support`

Это свойство подобно свойству `jQuery.browser`, но оно предназначено для переносимой проверки поддерживаемых возможностей (раздел 13.4.3) вместо менее надежного способа определения типа броузера. Значением свойства `jQuery.support` является объект, все свойства которого имеют логические значения и определяют наличие или отсутствие поддержки различных возможностей броузеров. Большинство свойств объекта `jQuery.support` содержат низкоуровневую информацию, используемую внутренними механизмами библиотеки jQuery. В основном они представляют интерес для разработчиков расширений и мало чем полезны прикладным программистам. Одно исключение – свойство `jQuery.support.boxModel`: оно имеет значение `true`, если броузер использует блочную модель «context-box», соответствующую стандарту CSS, и значение `false` в IE6 и IE7, работающих в режиме совместимости (раздел 16.2.3.1).

`jQuery.trim()`

Эта функция похожа на метод `trim()`, добавленный в строки стандартом ES5. Она принимает единственный строковый аргумент и возвращает его копию, из которой удалены начальные и завершающие пробельные символы.

19.8. Селекторы и методы выбора в библиотеке jQuery

На протяжении всей главы мы использовали функцию выбора `$()` из библиотеки jQuery, применяя простые CSS-селекторы. Теперь пришло время поближе познакомиться с грамматикой селекторов jQuery, а также с некоторыми методами, позволяющими фильтровать и расширять множество выбранных элементов.

19.8.1. Селекторы jQuery

Библиотека jQuery поддерживает достаточно полное подмножество селекторов, определяемых проектом стандарта «CSS3 Selectors», расширенное нестандартными, но очень удобными псевдоклассами. Основы CSS-селекторов обсуждались в разделе 15.2.5. Здесь мы продолжим это обсуждение и дополнительно познакомимся с более сложными селекторами. Имейте в виду, что в этом разделе описываются селекторы, реализованные в библиотеке jQuery. Многие из них могут использоваться в таблицах стилей CSS, но не все.

Грамматика селекторов делится на три уровня. Вы наверняка уже встречались с простейшими видами селекторов ранее. Селектор «#test» выбирает элемент с атрибутом `id`, имеющим значение «test». Селектор «blockquote» выбирает все элементы `<blockquote>` в документе, а селектор «div.note» выбирает все элементы `<div>` с атрибутом `class`, имеющим значение «note». Простые селекторы можно объединять в «комбинированные селекторы», такие как «div.note>p» и «blockquote i», отделяя их *символом-комбинатором*. И простые, и комбинированные селекторы можно группировать в списки, отделяя их точкой с запятой. Такие группы селекторов являются наиболее универсальной разновидностью селекторов, обычно передаваемых функции `$()`. Прежде чем перейти к обсуждению комбинированных селекторов и групп селекторов, необходимо познакомиться с синтаксисом простых селекторов.

19.8.1.1. Простые селекторы

Простой селектор начинается (явно или неявно) с имени тега. Если, к примеру, интерес представляют только элементы <p>, простой селектор должен начинаться с «p». Если требуется выбрать элементы независимо от имени тега, используется шаблонный символ «*». Если селектор не начинается с имени тега или шаблонного символа, подразумевается присутствие шаблонного символа.

Имя тега или шаблонный символ определяют начальное множество элементов документа, кандидатов на выбор. Фрагмент селектора, следующий за определением имени тега, состоит из нуля или более фильтров. Фильтры применяются слева направо, в порядке их следования, и каждый из них сужает множество выбранных элементов. Фильтры, поддерживаемые библиотекой jQuery, см. в табл. 19.1.

Обратите внимание, что некоторые из фильтров, перечисленных в табл. 19.1, принимают аргументы в круглых скобках. Следующий селектор, например, выберет абзацы, которые являются первыми или каждыми третьими дочерними элементами своих родителей при условии, что они содержат слово «JavaScript» и не содержат элемент <a>.

```
p:nth-child(3n+1):text(JavaScript):not(:has(a))
```

Обычно фильтры действуют более эффективно, если им предшествует имя тега. Например, вместо использования простого фильтра «:radio» для выбора радиокнопок лучше использовать селектор «input:radio». Исключение составляют фильтры, проверяющие значение атрибута id, которые наиболее эффективно действуют, когда они употребляются автономно. Например, селектор «#address» обычно действует эффективнее, чем более явный селектор «form#address».

Таблица 19.1. Фильтры селекторов, поддерживаемые библиотекой jQuery

Фильтр	Описание
#id	Соответствует элементам с атрибутом id, имеющим значение *id*. Допустимые HTML-документы никогда не имеют более одного элемента с одним и тем же значением в атрибуте id, поэтому этот фильтр обычно используется как самостоятельный селектор.
.class	Соответствует элементам с атрибутом class, значение которого (интерпретируется как список слов, разделенных пробелами) включает слово *class*.
[*attr*]	Соответствует элементам, имеющим атрибут *attr* (независимо от значения).
[*attr*=*val*]	Соответствует элементам, имеющим атрибут *attr* со значением *val*.
[*attr*!=*val*]	Соответствует элементам, не имеющим атрибут *attr*, или элементам с атрибутом *attr*, значение которого не равно *val* (расширение jQuery).
[*attr*^=*val*]	Соответствует элементам с атрибутом *attr*, значение которого начинается с *val*.
[*attr*$=*val*]	Соответствует элементам с атрибутом *attr*, значение которого оканчивается на *val*.
[*attr**=*val*]	Соответствует элементам с атрибутом *attr*, значение которого содержит *val*.
[*attr*~=*val*]	Соответствует элементам с атрибутом *attr*, когда элемент, интерпретируемый как список слов, разделенных пробелами, содержит слово *val*. То есть селектор «div.note» – это то же самое, что и «div[class~=note]».

Фильтр	Описание
[*attr*|=*val*]	Соответствует элементам с атрибутом *attr*, значение которого начинается с *val* и, возможно, следующим за ним дефисом и любыми другими символами.
:animated	Соответствует элементам, к которым в настоящее время применяется анимационный эффект jQuery.
:button	Соответствует элементам <button type="button"> и <input type="button"> (расширение jQuery).
:checkbox	Соответствует элементам <input type="checkbox"> (расширение jQuery). Эффективнее всего использовать этот фильтр с именем тега input: «input:checkbox».
:checked	Соответствует отмеченным элементам ввода.
:contains(*text*)	Соответствует элементам, содержащим указанный текст *text* (расширение jQuery). Текст в этом фильтре ограничивают круглые скобки, т. е. кавычки здесь не нужны. Текстовое содержимое элементов проверяется этим фильтром по значениям их свойств textContent и innerText; это простой, необработанный текст документа, из которого исключены теги и комментарии.
:disabled	Соответствует элементам в запрещенном состоянии.
:empty	Соответствует элементам, не имеющим потомков, в том числе текстовых узлов.
:enabled	Соответствует элементам в незапрещенном состоянии.
:eq(*n*)	Соответствует только *n*-му элементу в списке совпадений, расположенных в порядке следования в документе, отсчет в котором начинается с нуля (расширение jQuery).
:even	Соответствует элементам с четными индексами в списке. Поскольку первый элемент имеет индекс 0, фактически этот фильтр соответствует первому, третьему, пятому (и так далее) элементам (расширение jQuery).
:file	Соответствует элементам <input type="file"> (расширение jQuery).
:first	Соответствует только первому элементу в списке. То же, что и :eq(0) (расширение jQuery).
:first-child	Соответствует только элементам, которые являются первыми дочерними элементами своих родителей. Обратите внимание, что этот фильтр полностью отличается от фильтра :first.
:gt(*n*)	Соответствует элементам в списке совпадений, расположенным в порядке следования в документе, отсчет в котором начинается с нуля, чей индекс больше значения *n* (расширение jQuery).
:has(*sel*)	Соответствует элементам, имеющим потомков, соответствующих селектору *sel*.
:header	Соответствует любым элементам-заголовкам: <h1>, <h2>, <h3>, <h4>, <h5> или <h6> (расширение jQuery).
:hidden	Соответствует всем невидимым на экране элементам, т. е. элементам, значения свойств offsetWidth и offsetHeight которых равны 0.
:image	Соответствует элементам <input type="image">. Обратите внимание, что этот фильтр не соответствует элементам <img> (расширение jQuery).
:input	Соответствует элементам ввода: <input>, <textarea>, <select> и <button> (расширение jQuery).

Таблица 19.1 (продолжение)

Фильтр	**Описание**
`:last`	Соответствует последнему элементу в списке совпадений (расширение jQuery).
`:last-child`	Соответствует всем элементам, которые являются последними дочерними элементами своих родителей. Обратите внимание, что этот фильтр полностью отличается от фильтра `:last`.
`:lt(`*n*`)`	Соответствует элементам в списке совпадений, расположенным в порядке следования в документе, отсчет в котором начинается с нуля, чей индекс меньше значения *n* (расширение jQuery).
`:not(`*sel*`)`	Соответствует элементам, которые *не* соответствуют селектору *sel*.
`:nth(`*n*`)`	Синоним фильтра `:eq(`*n*`)` (расширение jQuery).
`:nth-child(`*n*`)`	Соответствует элементам, которые являются *n*-ми дочерними элементами своих родителей. *n* может быть числом, словом «even», словом «odd» или формулой. Для выбора второго, четвертого (и так далее) элементов в списках дочерних элементов их родителей можно использовать фильтр `:nth-child(even)`. Для выбора первого, третьего (и так далее) элементов можно использовать фильтр `:nth-child(odd)`. В самом общем случае *n* может быть формулой вида *xn* или *xn+y*, где *x* и *y* являются целыми числами, а *n* – символом `n`. То есть фильтр `nth-child(3n+1)` выберет первый, четвертый, седьмой (и так далее) элементы. Обратите внимание, что отсчет индексов в этом фильтре ведется с единицы, поэтому индекс первого дочернего элемента его родителя считается нечетным и он соответствует формуле `3n+1`, но не соответствует формуле `3n`. Сравните этот фильтр с фильтрами `:even` и `:odd`, которые начинают отсчет элементов в списке совпадений с нуля.
`:odd`	Соответствует элементам с нечетными индексами в списке (отсчет начинается с нуля). Обратите внимание, что элементы с индексами 1 и 3 являются вторым и четвертым элементами, соответственно (расширение jQuery).
`:only-child`	Соответствует элементам, являющимся единственными дочерними элементами своих родителей.
`:parent`	Соответствует элементам, которые являются родителями. Этот фильтр является обратным по отношению к фильтру `:empty` (расширение jQuery).
`:password`	Соответствует элементам `<input type="password">` (расширение jQuery).
`:radio`	Соответствует элементам `<input type="radio">` (расширение jQuery).
`:reset`	Соответствует элементам `<input type="reset">` и `<button type="reset">` (расширение jQuery).
`:selected`	Соответствует элементам `<option>`, которые были отмечены. Для выбора отмеченных флажков и переключателей (радиокнопок) используйте фильтр `:checked` (расширение jQuery).
`:submit`	Соответствует элементам `<input type="submit">` и `<button type="submit">` (расширение jQuery).
`:text`	Соответствует элементам `<input type="text">` (расширение jQuery).
`:visible`	Соответствует всем элементам, которые видимы в текущий момент, т. е. элементам, свойства `offsetWidth` и `offsetHeight` которых не равны нулю. Этот фильтр является обратным по отношению к фильтру `:hidden`.

19.8.1.2. Комбинированные селекторы

Простые селекторы могут объединяться с использованием специальных операторов, или «комбинаторов», для представления отношений между элементами в дереве документа. В табл. 19.2 перечислены комбинированные селекторы, поддерживаемые библиотекой jQuery. Эти же комбинированные селекторы поддерживаются стандартом CSS3.

Ниже приводится несколько примеров комбинированных селекторов:

```
"blockquote i"     // Соответствует элементу <i> в элементе <blockquote>
"ol > li"          // Элемент <li>, прямой потомок элемента <ol>
"#output + *"      // Братские элементы, следующие за элементом с id="output"
"div.note > h1 + p" // Элемент <p>, следующий за <h1> в <div class="note">
```

Обратите внимание, что комбинированные селекторы не ограничены комбинациями из двух селекторов: допустимыми являются также комбинации из трех и более селекторов. Обработка комбинированных селекторов выполняется слева направо.

Таблица 19.2. Комбинированные селекторы, поддерживаемые библиотекой jQuery

Комбинированный селектор	Описание
`A B`	Выбирает элементы документа, которые соответствуют селектору `B` и являются потомками элементов, соответствующих селектору `A`. Обратите внимание, что здесь символом-комбинатором является простой пробел.
`A > B`	Выбирает элементы документа, которые соответствуют селектору `B` и являются непосредственными потомками элементов, соответствующих селектору `A`.
`A + B`	Выбирает элементы документа, которые соответствуют селектору `B` и следуют непосредственно за элементами, соответствующими селектору `A` (текстовые узлы и комментарии в учет не принимаются).
`A ~ B`	Выбирает элементы документа, которые соответствуют селектору `B` и являются братскими для элементов, соответствующих селектору `A`.

19.8.1.3. Группы селекторов

Группа селекторов, которая является разновидностью селекторов, передаваемых функции `$()` (или используемых в таблицах стилей), – это просто список простых или комбинированных селекторов, разделенных запятыми. Группе селекторов соответствуют все элементы, которые соответствуют любому из комбинированных селекторов в группе. С позиции нашего обсуждения даже простой селектор можно рассматривать как комбинированный. Ниже приводится несколько примеров групп селекторов:

```
"h1, h2, h3"        // Соответствует элементам <h1>, <h2> и <h3>
"#p1, #p2, #p3"     // Соответствует элементам с id, равным p1, p2 и p3
"div.note, p.note"  // Соответствует элементам <div> и <p> с class="note"
"body>p,div.note>p" // <p>, вложенный в <body>, и <div class="note">
```

Обратите внимание, что синтаксис селекторов CSS и jQuery позволяет использовать круглые скобки в некоторых фильтрах в простых селекторах, но не допускает более обобщенного их использования для группировки селекторов. Нельзя поместить группу селекторов или комбинированный селектор в скобки и интерпретировать их как простой селектор, например:

```
(h1, h2, h3)+p      // Недопустимо
h1+p, h2+p, h3+p    // Следует использовать этот вариант
```

19.8.2. Методы выбора

В дополнение к грамматике селекторов, поддерживаемой функцией `$()`, библиотека jQuery определяет несколько методов выбора. Большинство методов из библиотеки jQuery, с которыми мы встречались в этой главе до сих пор, выполняют некоторые операции над выбранными элементами. Методы выбора отличаются от них: они изменяют множество выбранных элементов, фильтруя, расширяя или используя его в качестве отправной точки для нового выбора.

В данном разделе описываются эти методы выбора. Здесь вы увидите, что многие методы реализуют те же функциональные возможности, которые обеспечивает грамматика селекторов.

Самой простой является операция фильтрации выбора по позициям элементов в выбранном множестве. Метод `first()` возвращает объект `jQuery`, содержащий только первый выбранный элемент, а метод `last()` возвращает объект `jQuery`, содержащий только последний выбранный элемент. Более обобщенный метод `eq()` возвращает объект `jQuery`, содержащий единственный выбранный элемент с указанным индексом. (В версии jQuery 1.4 допускается использовать отрицательные индексы, в этом случае отсчет начинается с конца выбранного множества.) Обратите внимание, что эти методы возвращают объект `jQuery` с единственным элементом, что отличает их от обычной операции индексирования массива, которая возвращает единственный элемент без объекта `jQuery`, обертывающего его:

```
var paras = $("p");
paras.first()      // Выберет только первый элемент <p>
paras.last()       // Выберет только последний элемент <p>
paras.eq(1)        // Выберет второй элемент <p>
paras.eq(-2)       // Выберет второй с конца элемент <p>
paras[1]           // Сам второй элемент <p>
```

Универсальным методом фильтрации выбора по позиции является метод `slice()`. Этот метод действует подобно методу `Array.slice()`: он принимает начальный и конечный индексы (отрицательные индексы откладываются от конца массива) и возвращает объект `jQuery`, содержащий элементы с индексами от начального до конечного, не включая его. Если конечный индекс не указан, возвращаемый объект будет содержать все элементы от начального индекса и до конца:

```
$("p").slice(2,5)   // Выберет 3-й, 4-й и 5-й элементы <p>
$("div").slice(-3)  // Последние три элемента <div>
```

Метод `filter()` является универсальным методом фильтрации и может использоваться тремя разными способами:

- Если передать методу `filter()` строку с селектором, он вернет объект `jQuery`, содержащий только те из выбранных элементов, которые соответствуют этому селектору.
- Если передать методу `filter()` другой объект `jQuery`, он вернет новый объект `jQuery`, содержащий пересечение множеств элементов в двух исходных объек-

тах `jQuery`. Методу можно также передать массив элементов и даже единственный элемент документа.

- Если передать методу `filter()` функцию-предикат, эта функция будет вызвана для каждого выбранного элемента и метод `filter()` вернет объект `jQuery`, содержащий только те элементы, для которых функция-предикат вернет `true` (или любое другое истинное значение). Элемент будет передан функции в виде значения ссылки `this`, а индекс элемента – в виде аргумента. (Смотрите также описание метода `jQuery.grep()` в разделе 19.7.)

```
$("div").filter(".note")                          // Аналогично $("div.note")
$("div").filter($(".note"))                       // Аналогично $("div.note")
$("div").filter(function(idx) { return idx%2==0 }) // Аналогично $("div:even")
```

Метод `not()` действует точно так же, как метод `filter()`, за исключением того, что он инвертирует значение фильтра. Если передать методу `not()` строку с селектором, он вернет новый объект `jQuery`, содержащий только те из выбранных элементов, которые *не* соответствуют селектору. Если передать методу `not()` объект `jQuery`, массив элементов или единственный элемент, он вернет все выбранные элементы, кроме тех, что были явно указаны. Если передать методу `not()` функцию-предикат, он вызовет ее для каждого выбранного элемента, как и метод `filter()`, но вернет объект `jQuery`, содержащий только те выбранные элементы, для которых функция возвратит `false` или любое другое ложное значение:

```
$("div").not("#header, #footer");   // Все элементы <div>, кроме двух указанных
```

В версии jQuery 1.4 имеется метод `has()`, обеспечивающий еще один способ фильтрации выбора. Если ему передать селектор, он вернет новый объект `jQuery`, содержащий только те из выбранных элементов, которые имеют потомков, соответствующих селектору. Если передать методу `has()` элемент документа, он вернет только те выбранные элементы, которые являются предками указанного элемента:

```
$("p").has("a[href]")                // Абзацы, включающие ссылки
```

Метод `add()` не фильтрует и не сужает выбор, а расширяет его. Метод `add()` можно вызывать с любыми аргументами (кроме функций), которые можно передавать функции `$()`. Он возвращает первоначальное множество выбранных элементов, плюс элементы, которые были бы выбраны (или созданы) при передаче этих же аргументов функции `$()`. Метод `add()` удаляет повторные вхождения элементов и сортирует элементы в порядке их следования в документе:

```
// Эквивалентные способы выбора всех элементов <div> и <p>
$("div, p")                          // Используется группа селекторов
$("div").add("p")                    // Передать методу add() селектор
$("div").add($("p"))                 // Передать методу add() объект jQuery
var paras = document.getElementsByTagName("p"); // Объект, подобный массиву
$("div").add(paras);                 // Передать методу add() массив элементов
```

19.8.2.1. Использование результатов выбора в качестве контекста

Методы `filter()`, `add()` и `not()`, описанные выше, возвращают пересечение, объединение и разность независимых множеств выбранных элементов. Библиотека jQuery определяет также несколько других методов выбора, которые используют текущее множество выбранных элементов в качестве контекста. Для каждого выбранного элемента эти методы создают новое множество, используя выбранный элемент в качестве контекста, или отправной точки, и возвращают новый объект `jQuery`, содержащий объединение этих множеств. Как и метод `add()`, эти методы удаляют повторные вхождения одних и тех же элементов и сортируют их в порядке следования в документе.

Наиболее универсальным в этой категории методов выбора является метод `find()`. Он выполняет поиск потомков в каждом выбранном элементе, соответствующих указанной строке селектора, и возвращает новый объект `jQuery`, представляющий новое множество соответствующих потомков. Обратите внимание, что вновь выбранные элементы не объединяются с существующим множеством – они возвращаются в виде нового множества элементов. Отметьте также, что метод `find()` отличается от метода `filter()`, который просто сужает текущее множество выбранных элементов, не добавляя новых:

```
$("div").find("p")    // Отыскать элементы <p> в <div>. То же, что и $("div p")
```

Другие методы из этой категории возвращают новые объекты `jQuery`, представляющие дочерние, братские или родительские элементы для каждого выбранного элемента. Чаще всего им передается необязательный строковый аргумент с селектором. При вызове без селектора они возвращают, соответственно, все дочерние, братские или родительские элементы. При вызове с селектором они фильтруют список и возвращают только элементы, соответствующие селектору.

Метод `children()` возвращает дочерние элементы каждого выбранного элемента, фильтруя результат с применением необязательного селектора:

```
// Отыскать все элементы <span>, которые являются дочерними для элементов
// с id="header" и id="footer". Тот же результат дает вызов
// $("#header>span,#footer>span")
$("#header, #footer").children("span")
```

Метод `contents()` действует так же, как метод `children()`, но возвращает все дочерние узлы, включая текстовые, каждого выбранного элемента. Кроме того, для элементов `<iframe>` метод `contents()` возвращает объект `Document` с содержимым этого элемента. Обратите внимание, что метод `contents()` не принимает необязательный строковый аргумент с селектором – это обусловлено тем, что он возвращает объекты `Document`, не являющиеся элементами, а селекторы позволяют описывать только элементы.

Методы `next()` и `prev()` возвращают следующий и предыдущий братский элемент для каждого выбранного элемента, если они имеются. Если методу передать селектор, выбраны будут только братские узлы, соответствующие селектору:

```
$("h1").next("p")   // То же, что и $("h1+p")
$("h1").prev()      // Братские элементы перед элементами <h1>
```

Методы `nextAll()` и `prevAll()` возвращают все следующие и предыдущие братские элементы (если имеются) для каждого выбранного элемента. А метод `siblings()` возвращает все братские элементы для каждого выбранного элемента (элементы не считаются братскими по отношению к самим себе). Если любому из этих методов передать селектор, выбраны будут только братские узлы, соответствующие селектору:

```
$("#footer").nextAll("p")    // Все братские элементы <p>, следующие за #footer
$("#footer").prevAll()       // Все братские элементы, предшествующие #footer
```

В версии jQuery 1.4 и выше методы `nextUntil()` и `prevUntil()` принимают аргумент с селектором и выбирают все братские элементы, следующие за выбранным элементом или предшествующие ему, пока не будет встречен братский элемент, соответствующий селектору. При вызове без селектора они действуют точно так же, как методы `nextAll()` и `prevAll()` при вызове их без селектора.

Метод `parent()` возвращает родителя для каждого выбранного элемента:

```
$("li").parent()             // Родители элементов списка, такие как <ul> и <ol>
```

Метод `parents()` возвращает предков (вплоть до элемента `<html>`) для каждого выбранного элемента. Оба метода, `parent()` и `parents()`, принимают необязательный строковый аргумент с селектором:

```
$("a[href]").parents("p")    // Элементы <p>, содержащие ссылки
```

Метод `parentsUntil()` выбирает предков для каждого выбранного элемента, пока не будет встречен первый предок, соответствующий указанному селектору. Метод `closest()` принимает обязательный строковый аргумент с селектором и возвращает ближайшего предка (если имеется) для каждого выбранного элемента, соответствующего селектору. Этот метод рассматривает элементы как предки самим себе. В версии jQuery 1.4 методу `closest()` можно также передать второй необязательный аргумент, чтобы не дать библиотеке jQuery подняться по дереву предков выше указанного в этом аргументе элемента:

```
$("a[href]").closest("div")  // Самые внутренние элементы <div>, содержащие ссылки
$("a[href]").parentsUntil(":not(div)") // Все элементы <div>, непосредственно
                                       // обертывающие элементы <a>
```

19.8.2.2. Возврат к предыдущему выбору

Чтобы обеспечить возможность составления цепочек вызовов методов, большинство методов объекта `jQuery` возвращают объект, относительно которого они вызываются. Однако все методы, которые рассматривались в этом разделе, возвращают новые объекты `jQuery`. Они также могут включаться в цепочку вызовов, но вы должны иметь в виду, что методы, вызываемые в цепочке вслед за ними, будут оперировать другим множеством элементов, отличным от того, которое использовалось в начале цепочки.

Однако на самом деле ситуация несколько сложнее. Когда методы выбора, описанные здесь, создают и возвращают новый объект `jQuery`, они сохраняют в нем ссылку на прежний объект `jQuery`, на основе которого был порожден новый объект. В результате создается список, или стек объектов `jQuery`. Метод `end()` выталкивает объект на вершине этого стека и возвращает сохраненный объект `jQuery`. Вызов метода `end()` в цепочке восстанавливает множество выбранных элементов в предыдущее состояние. Взгляните на следующий фрагмент:

```
// Отыскать все элементы <div>, затем внутри множества выбранных элементов отыскать
// элементы <p>. Выделить элементы <p> цветом и затем окружить рамками элементы <div>.

// Сначала рассмотрим, как это сделать без цепочек вызовов методов
var divs = $("div");
var paras = divs.find("p");
```

```
paras.addClass("highlight");
divs.css("border", "solid black 1px");

// А теперь то же самое, составив цепочку вызовов методов
$("div").find("p").addClass("highlight").end().css("border", "solid black 1px");

// То же самое можно реализовать без вызова метода end(), просто переупорядочив операции
$("div").css("border", "solid black 1px").find("p").addClass("highlight");
```

Если вам когда-нибудь потребуется вручную определить множество выбранных элементов и обеспечить его совместимость с методом `end()`, передайте новое множество элементов в виде массива или в виде объекта, подобного массиву, методу `pushStack()`. Указанные элементы будут преобразованы в новое множество выбранных элементов, а предыдущее множество будет помещено в стек, откуда его можно будет извлечь вызовом метода `end()`:

```
var sel = $("div");                              // Выбрать все <div>
sel.pushStack(document.getElementsByTagName("p")); // Заменить его множеством
                                                 // всех элементов <p>
sel.end();                                    // Восстановить множество элементов <div>
```

Теперь, когда мы познакомились с методом `end()` и со стеком, хранящим множества выбранных элементов, нам осталось познакомиться с еще одним методом. Метод `andSelf()` возвращает новый объект `jQuery`, включающий все элементы из текущего множества выбранных элементов плюс все элементы (за исключением дубликатов) из предыдущего множества. Метод `andSelf()` действует подобно методу `add()` и для него больше подошло бы имя «addPrev». В качестве примера рассмотрим следующий вариант реализации предыдущего примера: он выделяет цветом элементы `<p>` и вмещающие их элементы `<div>` и затем добавляет рамки к элементам `<div>`:

```
$("div").find("p").andSelf().      // Отыскать <p> в <div> и объединить их
addClass("highlight").             // Выделить их все цветом
end().end().                       // Вытолкнуть со стека дважды до $("div")
css("border", "solid black 1px"); // Добавить рамки к элементам <div>
```

19.9. Расширение библиотеки jQuery с помощью модулей расширений

Библиотека `jQuery` написана так, что позволяет легко добавлять в нее новые функциональные возможности. Модули, добавляющие новые функциональные возможности, называются *расширениями* (*plug-in*), большое количество которых можно отыскать на сайте *http://plugins.jquery.com*. Расширения для библиотеки jQuery являются обычными файлами с программным кодом на языке JavaScript, и, чтобы задействовать их в своих веб-страницах, достаточно просто подключить их с помощью элемента `<script>`, как любую другую библиотеку на языке JavaScript (разумеется, расширения должны подключаться после подключения самой библиотеки jQuery).

Создание собственных расширений для библиотеки jQuery является почти тривиальной задачей. Вся хитрость заключается в объекте-прототипе `jQuery.fn`, который является прототипом для всех объектов `jQuery`. Если добавить новую функцию в этот объект, она превратится в метод объекта `jQuery`. Например:

```
jQuery.fn.println = function() {
    // Объединить все аргументы в одну строку, разделив их пробелами
    var msg = Array.prototype.join.call(arguments, " ");
    // Обойти в цикле все элементы в объекте jQuery
    this.each(function() {
        // В конец каждого из них добавить строку с простым текстом и <br/>.
        jQuery(this).append(document.createTextNode(msg)).append("<br/>");
    });
    // Вернуть объект jQuery, чтобы обеспечить возможность составления цепочек
    return this;
};
```

Определив эту функцию `jQuery.fn.println`, мы получаем возможность вызывать метод `println()` относительно любого объекта `jQuery`, как показано ниже:

```
$("#debug").println("x = ", x, "; y = ", y);
```

В `jQuery.fn` постоянно добавляются новые методы. Если обнаружится, что приходится «вручную» выполнять обход элементов в объекте jQuery с помощью метода `each()` и выполнять над ними некоторые операции, – это повод задуматься о необходимости реструктуризации программного кода, чтобы переместить вызов метода `each()` в дополнительный метод. Если при создании такого метода следовать приемам модульного программирования и соблюдать некоторые соглашения, принятые в библиотеке jQuery, этот дополнительный метод можно назвать расширением и поделиться им с другими. Ниже приводится перечень соглашений, которым необходимо следовать при создании расширений для библиотеки jQuery:

- Не полагайтесь на идентификатор $: подключающая страница может вызывать функцию `jQuery.noConflict()`, после чего `$()` уже не будет синонимом функции `jQuery()`. В коротких расширениях, как в примере выше, можно просто использовать имя `jQuery` вместо $. Если вы создаете большое расширение, то вы наверняка обернете его единственной анонимной функцией, чтобы избежать создания глобальных переменных. В этом случае можно использовать распространенный прием передачи ссылки на функцию `jQuery` в виде аргумента и принимать это значение в параметре с именем $:

```
(function($) { // Анонимная функция с одним параметром $
    // Здесь находится реализация расширения
}(jQuery));    // Вызвать функцию с объектом jQuery в виде аргумента
```

- Если метод расширения не должен возвращать какое-то свое значение, он должен возвращать объект `jQuery`, чтобы этот метод можно было использовать в цепочках вызовов. Обычно этот объект передается методам в виде ссылки `this`, которую можно просто вернуть вызывающей программе. Метод в примере выше завершается строкой `return this;`. Некоторые методы можно немного сократить (и сделать их сложнее для понимания), используя еще один распространенный прием: возвращая результат метода `each()`. Например, метод `println()` мог бы содержать программный код `return this.each(function() {...});`
- Если метод расширения принимает более двух параметров или параметров настройки, дайте пользователю метода передавать параметры в форме объекта (как мы видели на примере метода `animate()` в разделе 19.5.2 и функции `jQuery.ajax()` в разделе 19.6.3).

- Не засоряйте пространство имен jQuery лишними методами. Правильно оформленные расширения для библиотеки jQuery определяют минимальное количество методов, образуя непротиворечивый и удобный прикладной интерфейс. Обычно расширения jQuery определяют в объекте `jQuery.fn` единственный метод. Этот метод принимает в первом аргументе строку и интерпретирует ее как имя функции, которой следует передать остальные аргументы. Если расширение определяет единственный метод, его имя должно совпадать с именем расширения. Если необходимо определить более одного метода, в именах методов следует использовать имя расширения в качестве префикса.
- Если расширение привязывает обработчики событий, их следует поместить в пространство имен событий (раздел 19.4.4). В качестве имени пространства имен следует использовать имя расширения.
- Если расширение использует метод `data()` для связывания данных с элементами, все данные следует помещать в единственный объект и хранить его как единственное значение, дав ему имя, совпадающее с именем расширения.
- Файл с программным кодом расширения должен иметь имя в формате «jquery.plugin.js», где подстроку «plugin» следует заменить на имя расширения.

Расширения могут определять новые вспомогательные функции, добавляя их в сам объект `jQuery`. Например:

```
// Этот метод выводит свои аргументы (с помощью метода расширения println())
// в элемент с атрибутом id="debug". Если такой элемент отсутствует, он будет
// создан и добавлен в документ.
jQuery.debug = function() {
    var elt = jQuery("#debug");         // Отыскать элемент #debug
    if (elt.length == 0) {              // Создать, если он отсутствует
        elt = jQuery("<div id='debug'><h1>Debugging Output</h1></div>");
        jQuery(document.body).append(elt);
    }
    elt.println.apply(elt, arguments); // Вывести в него аргументы
};
```

Помимо создания новых методов можно также расширять и другие части библиотеки jQuery. В разделе 19.5, например, мы узнали, что имеется возможность добавлять новые имена, определяющие продолжительность визуальных эффектов (вдобавок к «fast» и «slow»), создавая новые свойства в объекте `jQuery.fx.speeds`, и добавлять новые функции переходов, включая их в объект `jQuery.easing`. Более того, с помощью расширений можно даже добавлять новые возможности в механизм селекторов библиотеки jQuery! Например, можно определить новые псевдоклассы фильтров (такие как `:first` и `:input`), добавив свойства в объект `jQuery.expr[':']`. Ниже приводится пример определения нового фильтра `:draggable`, который возвращает только элементы с атрибутом `draggable=true`:

```
jQuery.expr[':'].draggable = function(e) { return e.draggable === true; };
```

Добавив этот фильтр, мы сможем выбирать доступные для буксировки изображения вызовом `$("img:draggable")` вместо более длинного `$("img[draggable=true]")`.

Как можно заметить в примере выше, функции фильтра передается элемент DOM – кандидат на выбор. Она должна вернуть `true`, если элемент соответствует фильтру, и `false` – в противном случае. Многие нестандартным фильтрам достаточно одного аргумента с элементом, но в действительности им передается четыре аргумента. Во втором аргументе передается целочисленный индекс, определяющий позицию элемента в массиве кандидатов. Этот массив передается в четвертом аргументе, но ваша функция фильтра не должна модифицировать его. В третьем аргументе передается весьма интересное значение: это массив результатов вызова метода.exec() объекта RegExp. В четвертом элементе этого массива (с индексом 3) хранится значение, переданное псевдоклассу фильтра в круглых скобках (если оно имеется). Из этого значения удаляются все скобки и кавычки, и остается только строка. Например, ниже показано, как можно было бы реализовать псевдокласс :data(x), возвращающий true только для элементов, имеющих атрибут data-x (раздел 15.4.3):

```
jQuery.expr[':'].data = function(element, index, match, array) {
    // Примечание: В IE версии 7 и ниже метод hasAttribute() отсутствует
    return element.hasAttribute("data-" + match[3]);
};
```

19.10. Библиотека jQuery UI

Функциональные возможности, поддерживаемые библиотекой jQuery, сосредоточены на методах для работы с деревом DOM, стилями CSS, обработчиками событий и поддержки архитектуры Ajax. Все вместе это является отличным фундаментом для построения высокоуровневых абстракций, таких как виджеты пользовательского интерфейса, которые предоставляет библиотека jQuery UI. Полный охват возможностей библиотеки jQuery UI выходит далеко за рамки этой книги, и все, что нам доступно, – это краткий обзор. Саму библиотеку и документацию к ней можно найти на сайте *http://jqueryui.com*.

Как следует из ее имени, библиотека jQuery UI определяет множество виджетов пользовательского интерфейса: поля ввода с функцией автодополнения, элементы выбора даты, многостраничные виджеты и вкладки для организации информации, движки и индикаторы хода выполнения операции для визуального представления числовых значений и модальные диалоги для срочного оповещения пользователя. В дополнение к этим виджетам библиотека jQuery UI реализует более универсальные «механизмы взаимодействий», позволяющие легко сделать любой элемент документа буксируемым, изменяющим размеры, выбираемым или сортируемым. Наконец, библиотека jQuery UI определяет множество новых методов визуальных эффектов (включая возможность изменять цвет) вдобавок к тем, что предлагаются самой библиотекой jQuery, и добавляет множество новых функций переходов.

Библиотеку jQuery UI можно представить как пакет расширений для библиотеки jQuery, помещенных в один файл. Чтобы воспользоваться библиотекой jQuery UI, достаточно просто подключить ее к веб-странице после подключения основной библиотеки jQuery. На странице Download (Загрузка), на сайте *http://jqueryui.com*, можно выбрать компоненты, которые предполагается использовать, и сконструировать собственную версию библиотеки, что поможет вам сократить время загрузки ваших страниц в сравнении со случаем использования полной версии библиотеки jQuery UI.

Библиотека jQuery UI поддерживает темы оформления, которые реализуются в виде файлов CSS. То есть помимо загрузки в ваши страницы программного кода библиотеки jQuery UI вам также придется подключить файл CSS с выбранной темой оформления. На сайте библиотеки jQuery UI имеется множество готовых тем оформления, а также страница «ThemeRoller», позволяющая настраивать и загружать собственные темы оформления.

Виджеты и механизмы взаимодействий в библиотеке jQuery UI оформлены в виде расширений, каждое из которых определяет единственный метод объекта `jQuery`. Обычно, когда такие методы применяются к существующим элементам документа, они преобразуют эти элементы в виджеты. Например, чтобы превратить текстовое поле ввода в виджет выбора даты, при щелчке на котором отображается календарик, достаточно просто вызвать метод `datepicker()`, как показано ниже:

```
// Превратить элементы <input> с атрибутом class="date" в виджеты выбора даты
$("input.date").datepicker();
```

Для полноценного использования виджета из библиотеки jQuery UI необходимо знать три вещи: его параметры настройки, его методы и его события. Все виджеты в библиотеке jQuery UI являются настраиваемыми, и некоторые из них имеют весьма значительное количество настроек. Поведение и внешний вид виджетов можно настраивать, передавая объект с параметрами (подобный объекту с параметрами, который передается методу `animate()`) методу виджета.

Виджеты в библиотеке jQuery UI обычно определяют хотя бы несколько «методов» взаимодействия с виджетом. Однако, чтобы избежать быстрого роста количества методов объекта `jQuery`, виджеты в библиотеке jQuery UI определяют свои «методы» не как настоящие методы. Каждый виджет имеет всего один метод (такой как метод `datepicker()` в примере выше). Когда необходимо вызвать «метод» виджета, имя требуемого «метода» передается единственному настоящему методу, определяемому виджетом. Например, чтобы перевести виджет выбора даты в состояние запрещения, не нужно вызывать его метод `disableDatepicker()`; вместо этого нужно вызвать `datepicker("disable")`.

Как правило, виджеты в библиотеке jQuery UI определяют собственные события, которые генерируются в ответ на действия пользователя. Установить обработчики этих событий можно с помощью обычного метода `bind()` или с помощью свойств обработчиков событий объекта с параметрами, который передается методу виджета. В первом аргументе этим обработчикам, как обычно, передается объект `Event`. Некоторые виджеты во втором аргументе передают обработчикам объект «пользовательского интерфейса». Этот объект, как правило, хранит информацию о состоянии виджета.

Обратите внимание, что в документации к библиотеке jQuery UI иногда описываются «события», которые не являются настоящими событиями и являются скорее функциями обратного вызова, которые устанавливаются посредством объекта с параметрами настройки. Например, виджет выбора даты поддерживает несколько функций обратного вызова, которые вызываются им в различных ситуациях. Однако ни одна из этих функций не имеет сигнатуры, свойственной стандартным обработчикам событий, и обработчики этих «событий» нельзя зарегистрировать с помощью метода `bind()`. Вместо этого соответствующие функции обратного вызова необходимо указывать при настройке виджета в первом вызове метода `datepicker()`.

20

Сохранение данных на стороне клиента

Веб-приложения могут использовать прикладные программные интерфейсы броузеров для сохранения данных локально, на компьютере пользователя. Этот механизм сохранения данных на стороне клиента выполняет роль памяти для веб-броузеров. Веб-приложения могут сохранять, например, настройки пользователя или даже полную информацию о своем состоянии, чтобы иметь возможность возобновить работу точно с того момента, на котором их работа была прервана при последнем посещении. Хранилище на стороне клиента разграничивает данные в соответствии с происхождением, поэтому страницы с одного сайта не смогут читать данные, сохраненные страницами с другого сайта. Но две страницы с одного и того же сайта смогут использовать хранилище в качестве механизма взаимодействий. Например, данные, введенные в форме на одной странице, можно отображать в таблице на другой странице. Веб-приложения могут устанавливать срок хранения своих данных: данные могут храниться временно, т. е. получить такие данные из хранилища можно, пока не будет закрыто окно или пока броузер не завершит работу, или сохраняться на жестком диске и храниться постоянно, чтобы их можно было получить месяцы или даже годы спустя.

Существует несколько разновидностей хранилищ на стороне клиента:

Web Storage

Web Storage – это прикладной программный интерфейс, определение которого первоначально было частью стандарта HTML5, но впоследствии было выделено в отдельную спецификацию. Эта спецификация все еще находится в стадии проекта, но частично (переносимым образом) реализована во всех текущих броузерах, включая IE8. Этот прикладной интерфейс содержит объекты `localStorage` и `sessionStorage`, которые, по сути, являются постоянно хранимыми ассоциативными массивами, отображающими ключи в строковые значения. Интерфейс Web Storage очень прост в использовании, он подходит для хранения больших (но не огромных) объемов данных и доступен во всех текущих броузерах, но не поддерживается старыми броузерами. Объекты `localStorage` и `sessionStorage` описываются в разделе 20.1.

Cookies

Cookies – старейший механизм хранения данных на стороне клиента, который предназначен для использования серверными сценариями. В языке JavaScript имеется довольно неудобный прикладной интерфейс, позволяющий управлять cookies на стороне клиента, но этот механизм сложен в использовании и подходит лишь для хранения небольших объемов текстовых данных. Кроме того, любые данные, хранящиеся в виде cookies, всегда передаются серверу с каждым HTTP-запросом, даже если эти данные представляют интерес только для клиента. Однако механизм cookies по-прежнему представляет определенный интерес для разработчиков клиентских сценариев, потому что он поддерживается всеми броузерами, старыми и новыми. Однако когда механизм Web Storage получит более широкое распространение, механизм cookies вернется к своей первоначальной роли в качестве механизма хранения данных для серверных сценариев на стороне клиента. Механизм cookies рассматривается в разделе 20.2.

IE User Data

Корпорация Microsoft реализовала в IE версии 5 и выше свой собственный механизм хранения данных на стороне клиента, известный как «userData». Механизм userData позволяет хранить достаточно большие объемы строковых данных и может использоваться как альтернатива механизму Web Storage в IE версии 7 и ниже. Прикладной интерфейс механизма userData рассматривается в разделе 20.3.

Offline Web Applications

Стандарт HTML5 определяет прикладной программный интерфейс «Offline Web Applications» (автономные веб-приложения), позволяющий кэшировать веб-страницы и связанные с ними ресурсы (сценарии, CSS-файлы, изображения и т. д.). Это хранилище предназначено для сохранения веб-приложений целиком, а не только их данных, и позволяет веб-приложениям устанавливать себя, давая возможность использовать их даже при отсутствии соединения с Интернетом. Автономные веб-приложения рассматриваются в разделе 20.4.

Базы данных для Веб

Разработчики, которым приходится работать с по-настоящему огромными объемами данных, предпочитают использовать базы данных, и многие производители начинают включать в свои броузеры функциональные возможности доступа к базам данных на стороне клиента. Броузеры Safari, Chrome и Opera включают прикладной интерфейс к базе данных SQL. Однако попытка стандартизации этого прикладного интерфейса потерпела неудачу, и весьма маловероятно, что он будет реализован в Firefox и IE. Существует альтернативный прикладной интерфейс доступа к базам данных, который был стандартизован под названием «Indexed Database API». Это прикладной интерфейс к простейшей объектной базе данных, не поддерживающей язык запросов. Оба прикладных интерфейса являются асинхронными и требуют использования обработчиков событий, что усложняет их использование. Они не будут описываться в этой главе, но в разделе 22.8 вы найдете краткий обзор и пример применения механизма IndexedDB API.

Прикладной интерфейс к файловой системе

В главе 18 было показано, что современные броузеры поддерживают объект `File`, позволяющий с помощью объекта `XMLHttpRequest` выгружать файлы, выбранные пользователем. Проекты родственных стандартов определяют прикладной интерфейс для организации частной, локальной файловой системы и выполнения операций чтения и записи в файлы, находящиеся в этой файловой системе. Эти появляющиеся прикладные интерфейсы описываются в разделе 22.7. Когда они получат более широкое распространение, у веб-приложений будет возможность использовать механизмы хранения данных, основанные на файлах, уже знакомые многим программистам.

Сохранность, безопасность и конфиденциальность

Веб-броузеры часто предлагают пользователям сохранить пароли и сохраняют их на диске в зашифрованном виде. Но ни один из механизмов хранения данных на стороне клиента, описываемый в этой главе, никак не связан с шифрованием: все, что вы будете сохранять, будет сохраняться на жестком диске в незашифрованном виде. То есть хранящиеся данные могут быть извлечены чересчур любопытным пользователем, имеющим доступ к компьютеру, или злонамеренным программным обеспечением (например, разнообразными шпионскими программами), находящимся на компьютере. По этой причине ни один из механизмов хранения данных на стороне клиента никогда не должен использоваться для хранения паролей, номеров банковских счетов или другой конфиденциальной информации. Запомните: тот факт, что пользователь вводит какую-то информацию в поля форм при взаимодействии с вашим веб-сайтом, еще не означает, что он хочет сохранить копию этой информации на диске. Возьмите в качестве примера номер кредитной карты. Это конфиденциальная информация, которую люди предпочитают сохранять в тайне в своих бумажниках. Сохранить эту информацию с помощью механизма хранения данных на стороне клиента – это все равно что написать номер кредитной карты на бумажке и приклеить ее к клавиатуре.

Кроме того, учтите, что многие пользователи не доверяют веб-сайтам, использующим cookies или другие механизмы хранения данных на стороне клиента для целей, которые напоминают «слежение». Применяйте механизмы хранения, описываемые в этой главе, для повышения удобства работы с вашим сайтом, но не используйте их как механизм сбора конфиденциальной информации. Если появится слишком большое количество сайтов, дискредитирующих механизмы хранения данных на стороне клиента, пользователи будут отключать их или часто очищать хранилища, что сделает невозможным и их использование, и работу сайтов, опирающихся на их применение.

20.1. Объекты localStorage и sessionStorage

Броузеры, реализующие положения проекта спецификации «Web Storage», определяют в объекте Window два свойства: localStorage и sessionStorage. Оба свойства ссылаются на объект Storage – постоянно хранимый ассоциативный массив, отображающий строковые ключи в строковые значения. Объекты Storage действуют подобно обычным объектам в языке JavaScript: достаточно просто присвоить свойству объекта строку, и броузер автоматически сохранит ее. Разница между localStorage и sessionStorage заключается лишь в сроке хранения и области видимости: они определяют, как долго будут храниться данные и кому они будут доступны.

Ниже мы подробнее поговорим о сроке хранения и области видимости. А пока рассмотрим несколько примеров. Следующий фрагмент использует свойство localStorage, но он точно так же мог бы работать и со свойством sessionStorage:

```
var name = localStorage.username;        // Получить сохраненное значение.
name = localStorage["username"];         // Эквивалентная форма обращения, как к массиву
if (!name) {
    name = prompt("Как вас зовут?");     // Задать пользователю вопрос.
    localStorage.username = name;        // Сохранить ответ.
}

// Выполнить итерации по всем хранящимся парам имя/значение
for(var name in localStorage) {          // Итерации по всем хранящимся именам
    var value = localStorage[name];      // Получить значение для каждого из них
}
```

Объекты Storage также определяют методы для сохранения, извлечения и удаления данных. Эти методы рассматриваются в разделе 20.1.2.

Проект спецификации «Web Storage» определяет возможность сохранения структурированных данных (объектов и массивов), а также простых значений и данных встроенных типов, таких как даты, регулярные выражения и даже объекты File. Однако на момент написания этих строк броузеры позволяли сохранять только строки. Если потребуется сохранять и извлекать данные других типов, их можно кодировать и декодировать вручную. Например:

```
// При сохранении числа оно автоматически преобразуется в строку.
// Не забудьте выполнить обратное преобразование при извлечении из хранилища.
localStorage.x = 10;
var x = parseInt(localStorage.x);

// Преобразовать объект Date в строку при записи и обратно - при чтении
localStorage.lastRead = (new Date()).toUTCString();
var lastRead = new Date(Date.parse(localStorage.lastRead));

// Для кодирования любых простых или структурированных данных удобно
// использовать формат JSON
localStorage.data = JSON.stringify(data);    // Закодировать и сохранить
var data = JSON.parse(localStorage.data);    // Извлечь и декодировать.
```

20.1.1. Срок хранения и область видимости

Объекты localStorage и sessionStorage отличаются сроком хранения данных и областью видимости хранилища. Объект localStorage представляет долговременное хранилище данных: срок хранения не ограничен, и данные сохраняются на компьютере пользователя, пока не будут удалены веб-приложением или пока пользователь не потребует от броузера (посредством некоторого пользовательского интерфейса, предоставляемого броузером) удалить их.

Доступность данных в объекте `localStorage` ограничивается происхождением документа. Как описывалось в разделе 13.6.2, происхождение документа определяется такими параметрами, как протокол, имя хоста и номер порта, поэтому все следующие URL-адреса ссылаются на документы с разным происхождением:

```
http://www.example.com      // Протокол: http; имя хоста: www.example.com
https://www.example.com     // Другой протокол
http://static.example.com   // Другое имя хоста
http://www.example.com:8000 // Другой порт
```

Все документы, имеющие одно и то же происхождение, будут совместно использовать одни и те же данные в объекте `localStorage` (независимо от происхождения сценария, который фактически обращается к хранилищу `localStorage`). Они смогут читать и изменять данные друг друга. Но документы с разными происхождениями никогда не смогут прочитать или изменить данные друг друга (даже если оба они будут выполнять сценарий, полученный с одного и того же стороннего сервера).

Обратите внимание, что видимость данных в хранилище `localStorage` также ограничивается конкретным броузером. Если посетить сайт с помощью Firefox, а затем вновь посетить его, например, с помощью Chrome, никакие данные, сохраненные при первом посещении, не будут доступны при втором посещении.

Данные, сохраняемые в `sessionStorage`, имеют другой срок хранения: они хранятся, пока остается открытым окно верхнего уровня или вкладка броузера, в которой выполнялся сценарий, сохранивший эти данные. При закрытии окна или вкладки все данные, хранящиеся в `sessionStorage`, удаляются. (Отметьте, однако, что современные броузеры имеют возможность повторно открывать недавно закрытые вкладки и восстанавливать последний сеанс работы с броузером, поэтому срок хранения информации об этих вкладках и связанных с ними хранилищах `sessionStorage` может оказаться больше, чем кажется.)

Доступность данных в хранилище `sessionStorage`, как и в хранилище `localStorage`, ограничивается происхождением документа, т. е. документы с разным происхождением никогда не смогут совместно использовать одни и те же данные в `sessionStorage`. Но помимо этого доступность данных в хранилище `sessionStorage` ограничивается также окном. Если пользователь откроет в броузере две вкладки, отображающие документы с общим происхождением, эти две вкладки будут владеть разными хранилищами `sessionStorage`. Сценарий, выполняющийся в одной вкладке, не сможет прочитать или изменить данные, сохраненные в другой вкладке, даже если в обеих вкладках будет открыта одна и та же страница и будет выполняться один и тот же сценарий. Обратите внимание, что разграничение хранилищ `sessionStorage` в разных окнах касается только окон верхнего уровня. Если в одной вкладке броузера будет находиться несколько элементов `<iframe>` и в этих фреймах будут отображаться документы с общим происхождением, они будут совместно использовать одно и то же хранилище `sessionStorage`.

20.1.2. Прикладной программный интерфейс объекта Storage

Объекты `localStorage` и `sessionStorage` часто используются как обычные объекты языка JavaScript: присваивание значения свойству приводит к сохранению строки, а чтение свойства – к ее извлечению из хранилища. Но эти объекты определяют также более формальный прикладной интерфейс, основанный на методах. Сохранить значение можно с помощью метода `setItem()`, передав ему имя и значение. Извлечь значение можно с помощью метода `getItem()`, передав ему имя. Удалить значение можно с помощью метода `removeItem()`, передав ему имя. (В большинстве броузеров удалить значение можно также с помощью оператора `delete`, как если бы оно было обычным объектом, но этот прием не работает в IE8.) Удалить все хранящиеся значения можно вызовом метода `clear()` (без аргументов). Наконец, перечислить имена всех хранящихся значений можно с помощью свойства `length` и метода `key()`, передавая ему значения от `0` до `length-1`. Ниже приводится несколько примеров использования объекта `localStorage`. Этот программный код с тем же успехом мог бы использовать объект `sessionStorage`:

```
localStorage.setItem("x", 1); // Сохранить число под именем "x"
localStorage.getItem("x");    // Извлечь значение

// Перечислить все хранящиеся пары имя-значение
for(var i = 0; i < localStorage.length; i++) { // length дает количество пар
    var name = localStorage.key(i);            // Получить имя i-й пары
    var value = localStorage.getItem(name);    // Получить значение этой пары
}

localStorage.removeItem("x"); // Удалить элемент "x"
localStorage.clear();         // Удалить все остальные элементы
```

Несмотря на то что обычно удобнее сохранять и извлекать данные, обращаясь к свойствам, тем не менее иногда может потребоваться использовать эти методы. Во-первых, метод `clear()` не имеет эквивалента и является единственным способом удаления всех пар имя/значение в объекте `Storage`. Аналогично метод `removeItem()` является единственным переносимым способом удаления одной пары имя/значение, потому что IE8 не позволяет использовать оператор `delete` для этой цели.

Если производители броузеров реализуют все положения спецификации и позволят сохранять в объекте `Storage` другие объекты и массивы, появится еще одна причина использовать методы, такие как `setItem()` и `getItem()`. Объекты и массивы являются изменяемыми значениями, поэтому объекту `Storage` необходимо будет создавать копию сохраняемого значения, чтобы все последующие изменения в оригинале не коснулись хранящегося значения. Точно так же объект `Storage` должен будет создавать копию и при извлечении значения, чтобы никакие изменения в извлеченном значении не коснулись хранящегося значения. Когда такое копирование будет реализовано, использование интерфейса свойств может показаться запутывающим. Взгляните на следующий (гипотетический, пока броузеры не поддерживают возможность сохранения структурированных значений) фрагмент:

```
localStorage.o = {x:1}; // Сохранить объект, имеющий свойство x
localStorage.o.x = 2;   // Попытка установить свойство хранящегося объекта
localStorage.o.x        // => 1: свойство x не изменилось
```

Во второй строке в примере выше выполняется безуспешная попытка присвоить новое значение свойству хранящегося объекта, потому что интерпретатор сначала извлечет копию хранящегося объекта, присвоит свойству копии новое значение и затем удалит копию. В результате изменения не коснутся хранящегося объекта. Меньше шансов ошибиться, если использовать метод `getItem()`:

```
localStorage.getItem("o").x = 2; // Не предполагает сохранение значения 2
```

Наконец, еще одна причина отдать предпочтение более явному прикладному интерфейсу на основе методов заключается в возможности имитировать этот прикладной интерфейс поверх других механизмов сохранения данных в броузерах, которые пока не поддерживают спецификацию «Web Storage». В следующих разделах будут представлены примеры реализации интерфейса объекта `Storage` с применением cookies и userData в IE. При использовании прикладного интерфейса на основе методов можно писать программный код, который будет использовать свойство `localStorage`, когда оно доступно, и возвращаться к использованию других механизмов в противном случае. Такой программный код мог бы начинаться следующими строками:

```
// Определить, какой механизм хранения будет использоваться
var memory = window.localStorage ||
             (window.UserDataStorage && new UserDataStorage()) ||
             new CookieStorage();
// Затем отыскать требуемый элемент в хранилище
var username = memory.getItem("username");
```

20.1.3. События объекта Storage

При изменении данных, хранящихся в `localStorage` или `sessionStorage`, броузер генерирует событие «storage» во всех объектах `Window`, в которых доступны эти данные (но не в окне, где выполнялось сохранение). Если в броузере открыты две вкладки со страницами с общим происхождением и в одной из страниц производится сохранение значения в `localStorage`, в другой вкладке будет сгенерировано событие «storage». Не забывайте, что область видимости данных, хранящихся в `sessionStorage`, ограничивается окном верхнего уровня, поэтому при изменении данных в `sessionStorage` события «storage» будут генерироваться только при наличии нескольких фреймов. Обратите также внимание, что события «storage» генерируются, только когда содержимое хранилища действительно изменяется. Присваивание хранимому элементу его текущего значения, как и попытка удалить несуществующий элемент, не возбуждают событие.

Регистрация обработчиков события «storage» выполняется с помощью метода `addEventListener()` (или `attachEvent()` в IE). В большинстве броузеров для этой цели можно также использовать свойство `onstorage` объекта `Window`, но на момент написания этих строк данное свойство не поддерживалось в Firefox.

Объект события, связанный с событием «storage», имеет пять основных свойств (к сожалению, они не поддерживаются в IE8):

`key`

Имя или ключ сохраняемого или удаляемого элемента. Если был вызван метод `clear()`, это свойство будет иметь значение `null`.

`newValue`

Новое значение элемента или `null`, если был вызван метод `removeItem()`.

`oldValue`

Старое значение существующего элемента, изменившегося или удаленного, или значение `null`, если был создан новый элемент.

`storageArea`

Это свойство будет хранить значение свойства `localStorage` или `sessionStorage` целевого объекта `Window`.

`url`

URL-адрес (в виде строки) документа, сценарий которого выполнил операцию с хранилищем.

Наконец, обратите внимание, что объект `localStorage` и событие «storage» могут служить широковещательным механизмом, с помощью которого броузер может отправлять сообщения всем окнам, в которых в настоящий момент открыт один и тот же веб-сайт. Например, если пользователь потребует от веб-сайта прекратить воспроизводить анимационные эффекты, сценарий может сохранить соответствующий параметр настройки в `localStorage`, чтобы соблюсти это требование при последующих посещениях сайта. При сохранении параметра будет сгенерировано событие, что позволит другим окнам, отображающим тот же сайт, также удовлетворить это требование. В качестве другого примера представьте веб-приложение графического редактора, позволяющее пользователю отображать палитры с инструментами в отдельных окнах. При выборе пользователем некоторого инструмента приложение могло бы сохранять в `localStorage` признак выбранного инструмента и тем самым рассылать другим окнам извещения о том, что был выбран новый инструмент.

20.2. Cookies

Cookies – это небольшие фрагменты именованных данных, сохраняемые веб-броузером и связанные с определенными веб-страницами или веб-сайтами. Cookies первоначально предназначались для разработки серверных сценариев и на низшем уровне реализованы как расширение протокола HTTP. Данные в cookies автоматически передаются между веб-броузером и веб-сервером, благодаря чему серверные сценарии могут читать и записывать значения, сохраняемые на стороне клиента. В этом разделе будет показано, как клиентские сценарии могут работать с cookies, используя свойство `cookie` объекта `Document`.

Прикладной интерфейс для работы с cookies является одним из старейших, а это означает, что он поддерживается всеми броузерами. К сожалению, этот прикладной интерфейс слишком замысловат. В нем отсутствуют методы: операции чтения, записи и удаления cookies осуществляются с помощью свойства `cookie` объекта `Document` с применением строк специального формата. Срок хранения и область видимости можно указать отдельно для каждого cookie с помощью атрибутов. Эти атрибуты также определяются посредством записи строк специального формата в то же самое свойство `cookie`.

Почему «cookie»?

Особого смысла у термина «cookie» (булочка) нет, тем не менее появился он не «с потолка». В туманных анналах истории компьютеров термин «cookie», или «magic cookie», использовался для обозначения небольшой порции данных, в частности, привилегированных или секретных данных, вроде пароля, подтверждающих подлинность или разрешающих доступ. В JavaScript cookies применяются для сохранения информации о состоянии и могут служить средством идентификации для веб-броузера, хотя они не шифруются и не могут считаться безопасными (впрочем, это не относится к передаче их через защищенное соединение по протоколу `https:`).

В следующих подразделах сначала будут описаны атрибуты, которые определяют срок хранения и область видимости cookies, а затем будет продемонстрировано, как сохранять и извлекать значения cookies в сценариях на языке JavaScript. Завершится этот раздел примером реализации на основе cookies прикладного интерфейса, имитирующего объект `Storage`.

Как определить, когда поддержка cookies включена

Cookies пользуются дурной славой у многих пользователей Всемирной паутины из-за недобросовестного использования cookies, связанных не с самой веб-страницей, а с изображениями на ней. Например, сторонние cookies позволяют компаниям, предоставляющим услуги рекламного характера, отслеживать перемещение пользователей с одного сайта на другой, что вынуждает многих пользователей по соображениям безопасности отключать режим сохранения cookies в своих веб-броузерах. Поэтому, прежде чем использовать cookies в сценариях JavaScript, следует проверить, не отключен ли режим их сохранения. В большинстве броузеров это можно сделать, проверив свойство `navigator.cookieEnabled`. Если оно содержит значение `true`, значит, работа с cookies разрешена, а если `false` – запрещена (хотя при этом может быть разрешено использование временных cookies, срок хранения которых ограничивается продолжительностью сеанса работы броузера). Свойство `navigator.cookieEnabled` не является стандартным, поэтому если сценарий вдруг обнаружит, что оно не определено, придется проверить, поддерживаются ли cookies, попытавшись записать, прочитать и удалить тестовый cookie, используя прием, описываемый ниже.

20.2.1. Атрибуты cookie: срок хранения и область видимости

Помимо имени и значения каждый cookie имеет необязательные атрибуты, управляющие сроком его хранения и областью видимости. По умолчанию cookies являются временными – их значения сохраняются на период сеанса веб-броузера и теряются при закрытии броузера. Обратите внимание, что этот срок хранения не совпадает со сроком хранения данных в `sessionStorage`: доступность cookies не ограничивается единственным окном, поэтому период их хранения по умолчанию совпадает с периодом работы процесса броузера, а не какого-то одного окна. Чтобы cookie сохранялся после окончания сеанса, необходимо сообщить броузеру, как долго (в секундах) он должен храниться, указав значение атрибута *max-age*. Если указать срок хранения, броузер сохранит cookie в локальном файле и удалит его только по истечении срока хранения.

Видимость cookie ограничивается происхождением документа, как и при использовании хранилищ `localStorage` и `sessionStorage`, а также строкой пути к документу. Область видимости cookie может регулироваться посредством атрибутов *path* и *domain*. По умолчанию cookie связывается с создавшей его веб-страницей и доступен этой странице, а также другим страницам из того же каталога или любых его подкаталогов. Если, например, веб-страница *http://www.example.com/catalog/index.html* создаст cookie, то этот cookie будет также видим страницам *http://www.example.com/catalog/order.html* и *http://www.example.com/catalog/widgets/index.html*, но невидим странице *http://www.example.com/about.html*.

Этого правила видимости, принятого по умолчанию, обычно вполне достаточно. Тем не менее иногда значения cookie требуется использовать на всем многостраничном веб-сайте независимо от того, какая страница создала cookie. Например, если пользователь ввел свой адрес в форму на одной странице, целесообразно было бы сохранить этот адрес как адрес по умолчанию. Тогда этим адресом можно будет воспользоваться при следующем посещении тем же пользователем этой страницы, а также при заполнении им совершенно другой формы на любой другой странице, где требуется ввести адрес, например для выставления счета. Для этого в cookie можно определить атрибут *path*. И тогда любая страница того же веб-сервера с URL-адресом, начинающимся с указанного значения, сможет использовать этот cookie. Например, если для cookie, установленного страницей *http://www.example.com/catalog/widgets/index.html*, в атрибуте *path* установлено значение «/catalog», этот cookie также будет виден для страницы *http://www.example.com/catalog/order.html*. А если атрибут *path* установлен в значение «/», то cookie будет видим для любой страницы на веб-сервере *http://www.example.com*.

Установка атрибута *path* в значение «/» определяет такую же область видимости cookie, как для хранилища `localStorage`, а также говорит о том, что броузер должен передавать имя и значение cookie на сервер при запросе любой веб-страницы с этого сайта. Имейте в виду, что атрибут *path* не должен восприниматься как своеобразный механизм управления доступом. Если веб-странице потребуется прочитать cookies, принадлежащие какой-то другой странице на том же веб-сайте, она может просто загрузить эту страницу в скрытый элемент `<iframe>` и прочитать все cookies, принадлежащие документу во фрейме. Политика общего происхождения (раздел 13.6.2) предотвращает подобное «подглядывание» за cookies, установленных веб-страницами с других сайтов, но оно считается вполне допустимым для документов с одного и того же сайта.

По умолчанию cookies доступны только страницам с общим происхождением. Однако большим веб-сайтам может потребоваться возможность совместного использования cookies несколькими поддоменами. Например, серверу *order.example.com* может потребоваться прочитать значения cookie, установленного сервером *catalog.example.com*. В этой ситуации поможет атрибут *domain*. Если cookie, созданный страницей с сервера *catalog.example.com*, имеет в атрибуте *path* значение «/», а в атрибуте *domain* – значение «.example.com», этот cookie будет доступен всем веб-страницам в поддоменах *catalog.example.com*, *orders.example.com* и в любых других поддоменах в домене *example.com*. Если атрибут *domain* не установлен, его значением по умолчанию будет имя веб-сервера, на котором находится страница. Обратите внимание, что в атрибут *domain* нельзя записать значение, отличающееся от домена вашего сервера.

Последний атрибут cookie – это логический атрибут с именем *secure*, определяющий, как значения cookie передаются по сети. По умолчанию cookie не защищен, т.е. передается по обычному незащищенному HTTP-соединению. Однако если cookie помечен как защищенный, он передается, только когда обмен между броузером и сервером организован по протоколу HTTPS или другому защищенному протоколу.

20.2.2. Сохранение cookies

Чтобы связать временное значение cookie с текущим документом, достаточно присвоить его свойству `cookie` строку следующего формата:

```
имя=значение
```

Например:

```
document.cookie = "version=" + encodeURIComponent(document.lastModified);
```

При следующем чтении свойства `cookie` сохраненная пара имя/значение будет включена в список cookies документа. Значения cookie не могут содержать точки с запятой, запятые или пробельные символы. По этой причине для кодирования значения перед сохранением его в cookie, возможно, потребуется использовать глобальную JavaScript-функцию `encodeURIComponent()`. В этом случае при чтении значения cookie надо будет вызвать соответствующую функцию `decodeURIComponent()`.

Записанный таким способом cookie сохраняется в течение сеанса работы веб-броузера, но теряется при его закрытии пользователем. Чтобы создать cookie, сохраняющийся между сеансами броузера, необходимо указать срок его хранения (в секундах) с помощью атрибута `max-age`. Это можно сделать, присвоив свойству `cookie` строку следующего формата:

```
имя=значение; max-age=число_секунд
```

Следующая функция устанавливает cookie с дополнительным атрибутом `max-age`:

```
// Сохраняет пару имя/значение в виде cookie, кодируя значение с помощью
// encodeURIComponent(), чтобы экранировать точки с запятой, запятые и пробелы.
// Если в параметре daysToLive передается число, атрибут max-age
// устанавливается так, что срок хранения cookie истекает через
// указанное число дней. Если передать значение 0, cookie будет удален.
function setCookie(name, value, daysToLive) {
    var cookie = name + "=" + encodeURIComponent(value);
    if (typeof daysToLive === "number")
        cookie += "; max-age=" + (daysToLive*60*60*24);
    document.cookie = cookie;
}
```

Аналогичным образом можно установить атрибуты `path`, `domain` **и** `secure`, **дописав к значению cookie строки следующего формата перед его записью в свойство** `cookie`**:**

```
; path=путь
; domain=домен
; secure
```

Чтобы изменить значение cookie, установите его значение снова, указав то же имя, путь, домен и новое значение. При изменении значения cookie можно также переопределить срок его хранения, указав новое значение в атрибуте `max-age`.

Чтобы удалить cookie, установите его снова, указав то же имя, путь, домен и любое произвольное (возможно пустое) значение, а в атрибут `max-age` **запишите 0.**

20.2.3. Чтение cookies

Когда свойство `cookie` **используется в JavaScript-выражении, возвращаемое им значение содержит все cookies, относящиеся к текущему документу. Эта строка представляет собой список пар** *имя = значение*, **разделенных точками с запятой и пробелами. Значение не включает какие-либо атрибуты, которые могли быть установлены для cookie. При работе со свойством** `document.cookie` **обычно приходится использовать метод** `split()`, **чтобы разбить его значение на отдельные пары имя/значение.**

После извлечения значения cookie из свойства `cookie` **его требуется интерпретировать, основываясь на том формате или кодировке, которые были указаны создателем cookie. Например, cookie можно передать функции** `decodeURIComponent()`, **а затем функции** `JSON.parse()`.

В примере 20.1 определяется функция `getCookie()`, **которая анализирует свойство** `document.cookie` **и возвращает объект, свойства которого соответствуют парам имя/значение всех cookies в документе.**

Пример 20.1. Анализ свойства document.cookie

```
// Возвращает cookies документа в виде объекта с парами имя/значение.
// Предполагается, что значения cookie кодируются с помощью
// функции encodeURIComponent().
function getCookies() {
    var cookies = {};           // Возвращаемый объект
    var all = document.cookie;  // Получить все cookies в одной строке
    if (all === "")             // Если получена пустая строка,
        return cookies;         // вернуть пустой объект
    var list = all.split("; "); // Разбить на пары имя/значение
    for(var i = 0; i < list.length; i++) { // Для каждого cookie
        var cookie = list[i];
        var p = cookie.indexOf("=");        // Отыскать первый знак =
        var name = cookie.substring(0,p);   // Получить имя cookie
        var value = cookie.substring(p+1);  // Получить значение cookie
        value = decodeURIComponent(value);  // Декодировать значение
        cookies[name] = value;              // Сохранить имя и значение в объекте
    }
    return cookies;
}
```

20.2.4. Ограничения cookies

Cookies предназначены для сохранения небольших объемов данных серверными сценариями, которые должны передаваться на сервер при обращении к каждому соответствующему URL-адресу. Стандарт, определяющий cookies, рекомендует производителям броузеров не ограничивать количество и размеры сохраняемых cookies, но броузеры не обязаны сохранять в сумме более 300 cookies, 20 cookies на один веб-сервер или по 4 Кбайт данных на один cookie (в этом ограничении учитываются и значение cookie, и его имя). На практике броузеры позволяют сохранять гораздо больше 300 cookies, но ограничение на размер 4 Кбайт для одного cookie в некоторых броузерах по-прежнему соблюдается.

20.2.5. Реализация хранилища на основе cookies

Пример 20.2 демонстрирует, как поверх cookies можно реализовать методы, имитирующие прикладной интерфейс объекта `Storage`. Передайте конструктору `CookieStorage()` желаемые значения атрибутов *max-age* и *path* и используйте получившийся объект подобно тому, как вы использовали бы `localStorage` или `sessionStorage`. Однако имейте в виду, что этот пример не реализует событие «storage» и не выполняет автоматическое сохранение или извлечение значений при записи или чтении свойств объекта `CookieStorage`.

Пример 20.2. Реализация интерфейса объекта Storage на основе cookies

```
/*
 * CookieStorage.js
 * Этот класс реализует прикладной интерфейс объекта Storage, на который ссылаются
 * свойства localStorage и sessionStorage, но поверх HTTP Cookies.
 */
function CookieStorage(maxage, path) { // Аргументы определяют срок хранения
                                       // и область видимости

    // Получить объект, хранящий все cookies
    var cookies = (function() {        // Функция getCookies(), реализованная выше
        var cookies = {};              // Возвращаемый объект
        var all = document.cookie;     // Получить все cookies в одной строке
        if (all === "")                // Если получена пустая строка
            return cookies;            // вернуть пустой объект
        var list = all.split("; ");    // Разбить на пары имя/значение
        for(var i = 0; i < list.length; i++) { // Для каждого cookie
            var cookie = list[i];
            var p = cookie.indexOf("=");       // Отыскать первый знак =
            var name = cookie.substring(0,p);  // Получить имя cookie
            var value = cookie.substring(p+1); // Получить значение cookie
            value = decodeURIComponent(value); // Декодировать значение
            cookies[name] = value;             // Сохранить имя и значение
        }
        return cookies;
    }());

    // Собрать имена cookies в массиве
    var keys = [];
    for(var key in cookies) keys.push(key);
    // Определить общедоступные свойства и методы Storage API

    // Количество хранящихся cookies
    this.length = keys.length;

    // Возвращает имя n-го cookie или null, если n вышло за диапазон индексов
    this.key = function(n) {
        if (n < 0 || n >= keys.length) return null;
        return keys[n];
    };

    // Возвращает значение указанного cookie или null.
    this.getItem = function(name) { return cookies[name] || null; };

    // Сохраняет значение
    this.setItem = function(key, value) {
        if (!(key in cookies)) { // Если cookie с таким именем не существует
            keys.push(key);      // Добавить ключ в массив ключей
            this.length++;       // И увеличить значение length
        }

        // Сохранить пару имя/значение в множестве cookies.
        cookies[key] = value;

        // Установить cookie.
        // Предварительно декодировать значение и создать строку
        // имя=кодированное-значение
        var cookie = key + "=" + encodeURIComponent(value);

        // Добавить в строку атрибуты cookie
        if (maxage) cookie += "; max-age=" + maxage;
        if (path) cookie += "; path=" + path;

        // Установить cookie с помощью свойства document.cookie
        document.cookie = cookie;
    };

    // Удаляет указанный cookie
    this.removeItem = function(key) {
        if (!(key in cookies)) return; // Если не существует, ничего не делать

        // Удалить cookie из внутреннего множества cookies
        delete cookies[key];

        // И удалить ключ из массива имен.
        // Это легко можно было бы сделать с помощью метода indexOf() массивов,
        // определяемого стандартом ES5.
        for(var i = 0; i < keys.length; i++) { // Цикл по всем ключам
            if (keys[i] === key) {             // При обнаружении ключа
                keys.splice(i,1);              // Удалить его из массива.
                break;
            }
        }
        this.length--;                         // Уменьшить значение length

        // Наконец фактически удалить cookie, присвоив ему пустое значение
        // и истекший срок хранения.
        document.cookie = key + "=; max-age=0";
    };
    // Удаляет все cookies
    this.clear = function() {
        // Обойти все ключи и удалить cookies
        for(var i = 0; i < keys.length; i++)
            document.cookie = keys[i] + "=; max-age=0";
        // Установить внутренние переменные в начальное состояние
        cookies = {};
        keys = [];
        this.length = 0;
    };
}
```

20.3. Механизм сохранения userData в IE

В IE версии 5 и ниже поддерживается механизм сохранения данных на стороне клиента, доступный в виде нестандартного свойства `behavior` элемента документа. Использовать его можно следующим образом:

```
var memory = document.createElement("div");         // Создать элемент
memory.id = "_memory";                              // Дать ему имя
memory.style.display = "none";                      // Не отображать его
memory.style.behavior = "url('#default#userData')"; // Присоединить свойство
document.body.appendChild(memory);                  // Добавить в документ
```

После того как для элемента будет определено поведение «userData», он получает новые методы `load()` и `save()`. Вызов метода `load()` загружает сохраненные данные. Этому методу можно передать строку, напоминающую имя файла и идентифицирующую конкретный пакет хранящихся данных. После загрузки данных пары имя/значение становятся доступны в виде атрибутов элемента, получить которые можно с помощью метода `getAttribute()`. Чтобы сохранить новые данные, необходимо установить атрибуты вызовом метода `setAttribute()` и затем вызвать метод `save()`. Удалить значение можно с помощью методов `removeAttribute()` и `save()`. Ниже приводится пример использования элемента `memory`, инициализированного выше:

```
memory.load("myStoredData");                    // Загрузить сохраненные данные
var name = memory.getAttribute("username");     // Получить элемент данных
if (!name) {                                    // Если он не был определен,
    name = prompt("Как вас зовут?);             // запросить у пользователя
    memory.setAtttribute("username", name);     // Установить как атрибут
    memory.save("myStoredData");                // И сохранить до следующего раза
}
```

По умолчанию данные, сохраняемые с помощью механизма userData, имеют неограниченный срок хранения и сохраняются до тех пор, пока явно не будут удалены. Однако с помощью свойства `expires` можно определить дату, когда истечет срок хранения данных. Например, в предыдущий фрагмент можно было бы добавить следующие строки, чтобы указать, что срок хранения данных истечет через 100 дней:

```
var now = (new Date()).getTime();                // Текущее время в миллисекундах
var expires = now + 100 * 24 * 60 * 60 * 1000;   // + 100 дней в миллисекундах
expires = new Date(expires).toUTCString();       // Преобразовать в строку
memory.expires = expires;                        // Установить срок хранения
```

Данные, сохраняемые с помощью механизма userData в IE, доступны только для документов, хранящихся в том же каталоге, что и оригинальный документ, сохранивший их. Это более ограниченная область видимости, чем у cookies, которые также доступны документам в подкаталогах оригинального каталога. В механизме userData отсутствует какой-либо эквивалент атрибутов *path* и *domain* в cookies, позволяющий расширить область видимости данных.

Механизм userData позволяет сохранять гораздо большие объемы данных, чем cookies, но меньшие, чем объекты `localStorage` и `sessionStorage`.

Пример 20.3 реализует методы `getItem()`, `setItem()` и `removeItem()` интерфейса `Storage` поверх механизма userData в IE. (Он не реализует такие методы, как `key()` и `clear()`, потому что механизм userData не предоставляет возможность выполнять итерации по всем хранящимся элементам.)

Пример 20.3. Частичная реализация интерфейса Storage на основе механизма userData в IE

```
function UserDataStorage(maxage) {
    // Создать элемент документа и установить в нем специальное
    // свойство behavior механизма userData, чтобы получить доступ
    // к методам save() и load().
    var memory = document.createElement("div");         // Создать элемент
    memory.style.display = "none";                      // Не отображать его
    memory.style.behavior = "url('#default#userData')"; // Присоединить свойство behavior
    document.body.appendChild(memory);                  // Добавить в документ

    // Если указано значение параметра maxage, хранить данные maxage секунд
    if (maxage) {
        var now = new Date().getTime();     // Текущее время
        var expires = now + maxage * 1000; // maxage секунд от текущего времени
        memory.expires = new Date(expires).toUTCString();
    }

    // Инициализировать хранилище, загрузив сохраненные значения.
    // Значение аргумента выбирается произвольно, но оно должно совпадать
    // со значением, переданным методу save()
    memory.load("UserDataStorage");      // Загрузить сохраненные данные

    this.getItem = function(key) {       // Загрузить значения атрибутов
        return memory.getAttribute(key) || null;
    };
    this.setItem = function(key, value) {
        memory.setAttribute(key,value); // Сохранить значения как атрибуты
        memory.save("UserDataStorage"); // Сохранять после любых изменений
    };

    this.removeItem = function(key) {
        memory.removeAttribute(key);    // Удалить сохраненные значения
        memory.save("UserDataStorage"); // Сохранить новое состояние
    };
}
```

Поскольку программный код из примера 20.3 будет работать только в IE, можно воспользоваться условными комментариями IE, чтобы предотвратить его загрузку в броузерах, отличных от IE:

```
<!--[if IE]>
<script src="UserDataStorage.js"></script>
<![endif]-->
```

20.4. Хранилище приложений и автономные веб-приложения

Стандарт HTML5 определяет новую особенность «кэш приложений» (application cache), которая может использоваться веб-приложениями для сохранения самих себя локально в броузере пользователя. Объекты `localStorage` и `sessionStorage` позволяют сохранять данные веб-приложений, тогда как кэш приложений позволяет сохранять сами приложения – все файлы (HTML, CSS, JavaScript, изображения и т. д.), необходимые для работы приложения. Кэш приложений отличается от обычного кэша веб-броузера: он не очищается, когда пользователь очищает обычный кэш. И кэшированные приложения не очищаются по признаку LRU (least-recently used – давно не используемые), как это может происходить в обычном кэше фиксированного размера. Приложения сохраняются в кэше не временно: они устанавливаются и могут оставаться в нем, пока не удалят себя сами или не будут удалены пользователем. В действительности, кэш приложений вообще не является кэшем – для него больше подошло бы название «хранилище приложений» (application storage).

Основная причина необходимости установки веб-приложений локально заключается в обеспечении их доступности при работе в автономном режиме (например, в самолете или когда сотовый телефон находится вне доступа к сети). Веб-приложения, способные работать автономно, устанавливают себя в кэш приложений, используют `localStorage` для сохранения своих данных и реализуют механизм синхронизации для передачи сохраненных данных при подключении к сети. Пример автономного веб-приложения мы увидим в разделе 20.4.3, но сначала нам необходимо узнать, как приложение может установить себя в кэш приложений.

20.4.1. Объявление кэшируемого приложения

Чтобы установить приложение в кэш приложений, необходимо создать файл объявления: файл, перечисляющий URL всех ресурсов, необходимых приложению. Затем нужно просто добавить ссылку на файл объявления в основную HTML-страницу приложения, определив атрибут `manifest` в теге `<html>`:

```
<!DOCTYPE HTML>
<html manifest="myapp.appcache">
<head>...</head>
<body>...</body>
</html>
```

Файлы объявлений должны начинаться со строки «CACHE MANIFEST». В следующих строках должны перечисляться URL-адреса кэшируемых ресурсов. Относительные URL-адреса откладываются относительно URL-адреса файла объявления. Пустые строки игнорируются. Строки, начинающиеся с символа `#`, являются комментариями и также игнорируются. Перед комментариями могут быть пробелы, но они не могут следовать в строке за какими-либо непробельными символами. Ниже приводится пример простого файла объявления:

```
CACHE MANIFEST
# Строка выше определяет тип файла. Данная строка является комментарием

# Следующие строки определяют ресурсы, необходимые для работы приложения
myapp.html
myapp.js
myapp.css
images/background.png
```

Этот файл объявления служит признаком приложения, устанавливаемого в кэш. Если веб-приложение содержит более одной веб-страницы (более одного HTML-файла, которые могут быть открыты пользователем), в каждой из этих страниц должен быть определен атрибут `<html manifest=>`, ссылающийся на файл объявления. Факт наличия во всех страницах ссылок на один и тот же файл объявления недвусмысленно говорит о том, что все они должны кэшироваться вместе как части одного и того же веб-приложения. Если в приложении имеется всего несколько HTML-страниц, их обычно перечисляют непосредственно в файле объявления. Однако это совершенно необязательно: все файлы, ссылающиеся на файл объявления, будут считаться частью веб-приложения и вместе с ним будут установлены в кэш.

Простой файл объявления, подобный тому, что показан выше, должен перечислять *все* ресурсы, необходимые веб-приложению. После загрузки веб-приложения в первый раз и установки его в кэш при последующих обращениях к нему оно будет загружаться из кэша. Когда приложение загружается из кэша, все необходимые ему ресурсы должны быть перечислены в файле объявления. Ресурсы, которые не были перечислены, не загружаются. Эта политика имитирует работу в автономном режиме. Если простое кэшированное приложение сможет запускаться из кэша, оно точно так же сможет запускаться, когда броузер работает в автономном режиме. Более сложные веб-приложения в общем случае не могут кэшировать каждый необходимый им ресурс отдельно. Но они тем не менее могут использовать кэш приложений, если они имеют более сложные объявления.

MIME-тип объявления кэшируемого приложения

По соглашению файлам объявлений кэшируемых приложений даются имена с расширением *.appcache*. Однако это всего лишь соглашение, а для фактической идентификации типа файла веб-сервер *должен* отправлять файл объявления с MIME-типом «text/cache-manifest». Если при отправке файла объявления сервер установит в заголовке «Content-Type» любой другой MIME-тип, приложение не будет установлено в кэш. Вам может потребоваться специально настроить свой веб-сервер на использование нужного MIME-типа, например, создав в Apache файл *.htaccess* в каталоге веб-приложения.

20.4.1.1. Сложные объявления

При запуске приложения из кэша загружаются только ресурсы, перечисленные в файле объявления. В примере файла объявления, представленном выше, URL-адреса ресурсов перечисляются по одному. В действительности, файлы объявлений имеют более сложный синтаксис, чем было показано в этом примере, и существует еще два способа перечисления ресурсов в файлах объявлений. Для идентификации типов записей в объявлении используются специальные строки-заголовки разделов. Простые записи, как те, что были показаны выше, помещаются в раздел «CACHE:», который является разделом по умолчанию. Два других раздела начинаются с заголовков «NETWORK:» и «FALLBACK:». (В файле объявления может быть любое количество разделов, и они могут следовать в любом порядке.)

Раздел «NETWORK:» определяет ресурсы, которые никогда не должны кэшироваться и всегда должны загружаться из сети. Здесь можно перечислить, например, URL-адреса серверных сценариев. URL-адреса в разделе «NETWORK:» в действительности являются префиксами URL-адресов. Все ресурсы, URL-адреса которых начинаются с этих префиксов, будут загружаться только из сети. Если броузер работает в автономном режиме, то попытки обратиться к таким ресурсам будут оканчиваться неудачей. В разделе «NETWORK:» допускается использовать шаблонный URL-адрес «*». В этом случае броузер будет пытаться загружать из сети все ресурсы, не упомянутые в объявлении. Это фактически отменяет правило, которое требует явно перечислять в файле объявления все ресурсы, необходимые кэшируемому приложению.

Записи в разделе «FALLBACK:» включают два URL-адреса в каждой строке. Ресурс, указанный во втором URL, загружается и сохраняется в кэше. Первый URL используется как префикс. Все URL-адреса, соответствующие этому префиксу, не кэшируются и при возможности загружаются из сети. Если попытка загрузить ресурс с таким URL-адресом терпит неудачу, вместо него будет использоваться кэшированный ресурс, определяемый вторым URL-адресом. Представьте веб-приложение, включающее несколько видеоруководств. Поскольку видеоролики имеют большой объем, они не подходят для сохранения в локальном кэше. Для работы в автономном режиме файл объявления мог бы предусматривать отображение вместо них текстовой справки.

Ниже приводится более сложный файл объявления кэшируемого приложения:

```
CACHE MANIFEST

CACHE:
myapp.html
myapp.css
myapp.js

FALLBACK:
videos/ offline_help.html

NETWORK:
cgi/
```

20.4.2. Обновление кэша

При запуске кэшированного веб-приложения все его файлы загружаются непосредственно из кэша. Если броузер подключен к сети, он также асинхронно проверит наличие изменений в файле объявления. Если он изменился, будут загружены и установлены в кэш приложения новый файл объявления и все файлы, на которые он ссылается. Обратите внимание, что броузер не проверяет наличие

изменений в кэшированных файлах – проверяется только файл объявления. Например, если вы изменили файл сценария на языке JavaScript и вам необходимо, чтобы ваше веб-приложение обновило свой кэш, вам следует обновить файл объявления. Поскольку список файлов, необходимых приложению, при этом не изменяется, проще всего добиться требуемого результата, изменив номер версии:

```
CACHE MANIFEST
# MyApp версия 1 (изменяйте этот номер, чтобы заставить броузеры повторно
# загрузить следующие файлы)
MyApp.html
MyApp.js
```

Аналогично, если потребуется, чтобы веб-приложение удалило себя из кэша приложений, следует удалить файл объявления на сервере, чтобы на запрос этого файла возвращался бы HTTP-ответ 404 «Not Found», и изменить HTML-файл или файлы, удалив из них ссылки на файл объявления.

Обратите внимание, что броузеры проверяют файл объявления и обновляют кэш асинхронно, после (или во время) загрузки копии приложения из кэша. Для простых веб-приложений это означает, что после обновления файла объявления пользователь должен дважды загрузить приложение, чтобы получить обновленную версию: в первый раз будет загружена старая версия из кэша, после чего произойдет обновление файлов в кэше, а во второй раз из кэша будет загружена новая версия.

В ходе обновления кэша броузер запускает множество событий, что дает возможность зарегистрировать их обработчики и извещать пользователя. Например:

```
applicationCache.onupdateready = function() {
    var reload = confirm("Доступна новая версия приложения, которая\n" +
                         "будет использована при следующем запуске.\n" +
                         "Хотите ли перезапустить ее сейчас?");
    if (reload) location.reload();
}
```

Обратите внимание, что этот обработчик событий регистрируется в объекте `ApplicationCache`**, на который ссылается свойство** `applicationCache` **объекта** `Window`**. Броузеры, поддерживающие кэш приложений, определяют это свойство. Помимо события «updateready», показанного выше, существует еще семь различных событий, имеющих отношение к кэшу приложений. В примере 20.4 демонстрируются простые обработчики, которые выводят сообщения, информирующие пользователя о ходе обновления кэша и о его текущем состоянии.**

Пример 20.4. Обработка событий кэша приложений

```
// Эту функцию используют все обработчики событий, реализованные ниже, и выводят
// с ее помощью сообщения, информирующие о состоянии кэша приложений.
// Поскольку все обработчики отображают сообщения таким способом, они
// возвращают false, чтобы отменить дальнейшее распространение события
// и предотвратить вывод сообщений самим броузером.
function status(msg) {
    // Вывести сообщение в элементе документа с id="statusline"
    document.getElementById("statusline").innerHTML = msg;
    console.log(msg);      // А также в консоли для отладки
}
```

```
// Каждый раз, когда приложение загружается, броузер проверяет файл объявления.
// В начале этого процесса первым всегда генерируется событие "checking".
window.applicationCache.onchecking = function() {
    status("Проверка наличия новой версии.");
    return false;
};

// Если файл объявления не изменился и приложение уже имеется в кэше,
// генерируется событие "noupdate" и процедура проверки заканчивается.
window.applicationCache.onnoupdate = function() {
    status("Версия приложения не изменилась.")
    return false;
};

// Если приложение отсутствует в кэше или если изменился файл объявления,
// броузер загрузит и поместит в кэш все, что перечислено в файле объявления.
// Событие "downloading" свидетельствует о начале этой процедуры загрузки.
window.applicationCache.ondownloading = function() {
    status("Загружается новая версия");
    window.progresscount = 0;  // Используется в обработчике "progress" ниже
    return false;
};

// В ходе загрузки периодически генерируются события "progress",
// обычно после загрузки каждого файла.
window.applicationCache.onprogress = function(e) {
    // Объект события должен соответствовать событию "progress" (подобному тому,
    // что используется XHR2), что позволяет вычислять процент выполнения,
    // но на всякий случай мы заведем счетчик количества вызовов.
    var progress = "";
    if (e && e.lengthComputable) // Событие "progress": вычислить процент
        progress = " " + Math.round(100*e.loaded/e.total) + "%"
    else                         // Иначе сообщить кол-во вызовов
        progress = " (" + ++progresscount + ")"

    status("Загружается новая версия" + progress);
    return false;
};

// Когда приложение впервые загружается в кэш, по окончании загрузки
// броузер сгенерирует событие "cached".
window.applicationCache.oncached = function() {
    status("Приложение загружено и установлено локально");
    return false;
};

// Когда обновляется приложение, находящееся в кэше, то по завершении загрузки
// броузер сгенерирует событие "updateready". Обратите внимание, что при этом
// пользователь по-прежнему будет работать со старой версией приложения.
window.applicationCache.onupdateready = function() {
    status("Была загружена новая версия приложения. Перезапустите его.");
    return false;
};

// Если броузер выполняется в автономном режиме и файл объявления не может
// быть проверен, генерируется событие "error". Это же событие генерируется,
// когда некэшированное приложение ссылается на отсутствующий файл объявления.
window.applicationCache.onerror = function() {
    status("Невозможно загрузить файл объявления " +
           "или сохранить приложение в кэш");
    return false;
};

// Если кэшированное приложение ссылается на несуществующий файл объявления,
// генерируется событие "obsolete" и приложение удаляется из кэша.
// В следующий раз приложение будет целиком загружаться из сети, а не из кэша.
window.applicationCache.onobsolete = function() {
    status("Это приложение больше не кэшируется. " +
           "Перезапустите его, чтобы получить последнюю версию из сети.");
    return false;
};
```

Всякий раз, когда загружается HTML-файл с атрибутом `manifest`, броузер генерирует событие «checking» и загружает из сети файл объявления. Вслед за событием «checking» в разных ситуациях генерируются разные события:

Нет обновлений

Если приложение уже находится в кэше и файл объявления не изменился, броузер генерирует событие «noupdate».

Есть обновления

Если приложение находится в кэше и обнаружено изменение файла объявления, броузер генерирует событие «downloading» и приступает к загрузке и кэшированию всех файлов, перечисленных в файле объявления. С началом процесса загрузки начинают генерироваться события «progress». А по окончании загрузки генерируется событие «updateready».

Первая загрузка нового приложения

Если приложение отсутствует в кэше, события «downloading» и «progress» генерируются, как и для случая обновления кэше, описанного выше. Однако по окончании первой загрузки броузер генерирует событие «cached», а не «updateready».

Броузер работает в автономном режиме

Если броузер работает в автономном режиме, он не имеет возможности проверить файл объявления и генерирует событие «error». Это же событие генерируется, когда приложение, отсутствующее в кэше, ссылается на отсутствующий файл объявления.

Файл объявления отсутствует

Если броузер подключен к сети и приложение уже установлено в кэш, но при попытке получить файл объявления сервер возвращает ошибку 404 «Not Found», генерируется событие «obsolete» и приложение удаляется из кэша.

Обратите внимание, что все эти события можно отменить. Обработчики в примере 20.4 возвращают значение `false`, чтобы отменить действия, предусмотренные для событий по умолчанию. Это предотвращает вывод броузером своих собственных сообщений. (На момент написания этих строк ни один броузер не выводил никаких сообщений.)

В качестве альтернативы обработчикам событий, приложение может также использовать свойство `applicationCache.status` и с его помощью определять состояние кэша. Это свойство может иметь шесть разных значений:

`ApplicationCache.UNCACHED` (0)

Это приложение не имеет атрибута `manifest`: оно не кэшируется.

`ApplicationCache.IDLE` (1)

Файл объявления проверен, и в кэше находится последняя версия приложения.

`ApplicationCache.CHECKING` (2)

Броузер проверяет файл объявления.

`ApplicationCache.DOWNLOADING` (3)

Броузер загружает и сохраняет в кэше файлы, указанные в файле объявления.

`ApplicationCache.UPDATEREADY` (4)

Была загружена и установлена в кэш новая версия приложения.

`ApplicationCache.OBSOLETE` (5)

Файл объявления отсутствует, и приложение будет удалено из кэша.

Объект `ApplicationCache` также определяет два метода. Метод `update()` явно запускает процедуру проверки наличия новой версии приложения. Он заставляет броузер выполнить проверку файла объявления (и сгенерировать сопутствующие события), как если бы приложение загружалось в первый раз.

Метод `swapCache()` немного сложнее. Напомню, что, когда броузер загрузит и сохранит в кэше обновленную версию приложения, пользователь по-прежнему будет работать со старой версией. Он увидит новую версию, только когда перезагрузит приложение. Если пользователь не сделает этого, старая версия должна работать вполне корректно. И отметьте, что старая версия может по-прежнему загружать ресурсы из кэша: она, например, может использовать объект `XMLHttpRequest` для загрузки файлов, и эти файлы будут извлекаться из старой версии кэша. То есть, вообще говоря, броузер должен сохранять старую версию кэша, пока пользователь не перезагрузит приложение.

Метод `swapCache()` сообщает броузеру, что он может удалить старый кэш и удовлетворять все последующие запросы из нового кэша. Имейте в виду, что это не приводит к перезагрузке приложения: HTML-файлы, изображения, сценарии и другие ресурсы, которые уже были загружены, останутся старыми. Но на все последующие запросы будут возвращаться ресурсы из нового кэша. Это может вызвать конфликт между версиями, и в общем случае использовать этот метод не рекомендуется, если только вы специально не предусмотрели такую возможность. Представьте, например, приложение, которое не делает ничего и отображает экранную заставку, пока броузер проверяет файл объявления. Когда приложение получает событие «noupdate», оно продолжает работу и загружает начальную страницу. Если приложение получает событие «downloading», оно отображает соответствующий индикатор хода выполнения операции, пока обновляется кэш. И когда приложение получает событие «updateready», оно вызывает метод `swapCache()` и затем загружает обновленную начальную страницу из свежей версии в кэше.

Обратите внимание, что вызывать метод `swapCache()` имеет смысл, только когда свойство `status` имеет значение `ApplicationCache.UPDATEREADY` или `ApplicationCache.OBSOLETE`. (Вызов метода `swapCache()`, когда свойство `status` имеет значение `OBSOLETE`, сразу же удалит старую версию кэша, и все ресурсы будут загружаться из сети.) Если вызвать метод `swapCache()`, когда свойство `status` имеет любое другое значение, это приведет к исключению.

20.4.3. Автономные веб-приложения

Автономными называют веб-приложения, которые устанавливаются в кэш приложений и благодаря этому остаются доступными всегда, даже когда броузер работает в автономном режиме. В простейших случаях – таких как часы или генераторы фракталов – приложение уже имеет все, что ему необходимо для автономной работы. Но в более сложных веб-приложениях бывает необходимо выгружать данные на сервер: даже простейшим игровым приложениям может потребоваться выгружать на сервер высшие достижения игрока. Приложения, которым необходимо выгружать данные на сервер, также могут быть автономными, если для сохранения своих данных они будут использовать объект `localStorage` и затем выгружать данные, как только сеть будет доступна. Реализация синхронизации данных между локальным хранилищем и сервером может оказаться самым сложным этапом подготовки веб-приложения к работе в автономном режиме, особенно когда пользователь может обращаться к нескольким источникам данных.

Чтобы выполняться в автономном режиме, веб-приложение должно иметь возможность выяснить, работает ли оно в автономном режиме, и определять моменты подключения и отключения от сети. Проверить режим работы броузера можно с помощью свойства `navigator.onLine`. А определить изменение состояния подключения можно, зарегистрировав обработчики событий «online» и «offline» в объекте `Window`.

Эта глава завершается простым примером автономного веб-приложения, демонстрирующим использование этих приемов. Приложение называется PermaNote – это простое приложение управления заметками, которое сохраняет текст, введенный пользователем, в объекте `localStorage` и выгружает его на сервер, когда соединение с Интернетом будет доступно.[1] Приложение PermaNote дает пользователю создать единственную заметку и игнорирует проблемы аутентификации и авторизации – предполагается, что сервер обладает некоторым механизмом, позволяющим ему отличать одного пользователя от другого без использования какой-либо страницы входа. Реализация приложения PermaNote состоит из трех файлов. В примере 20.5 приводится содержимое файла объявления. В нем перечислены остальные два файла и указывается, что URL «note» не должен кэшироваться: этот URL-адрес используется для чтения и записи заметки на сервере.

Пример 20.5. permanote.appcache

```
CACHE MANIFEST
# PermaNote v8
permanote.html
```

---

[1] Идея этого примера была подсказана приложением Halfnote, написанным Аароном Будманом (Aaron Boodman). Приложение Halfnote было одним из первых автономных веб-приложений.

```
permanote.js
NETWORK:
note
```

В примере 20.6 приводится второй файл приложения PermaNote: это HTML-файл, который реализует пользовательский интерфейс простого редактора. Он отображает элемент `<textarea>` с панелью инструментов вдоль верхнего края и строкой состояния для сообщений вдоль нижнего края. Обратите внимание, что тег `<html>` имеет атрибут `manifest`.

Пример 20.6. permanote.html

```
<!DOCTYPE HTML>
<html manifest="permanote.appcache">
  <head>
    <title>Редактор PermaNote</title>
    <script src="permanote.js"></script>
    <style>
    #editor { width: 100%; height: 250px; }
    #statusline { width: 100%; }
    </style>
  </head>
  <body>
    <div id="toolbar">
      <button id="savebutton" onclick="save()">Сохранить</button>
      <button onclick="sync()">Синхронизировать</button>
      <button onclick="applicationCache.update()">Обновить приложение</button>
    </div>
    <textarea id="editor"></textarea>
    <div id="statusline"></div>
  </body>
</html>
```

Наконец, в примере 20.7 приводится сценарий на языке JavaScript, который обеспечивает работу веб-приложения PermaNote. Он определяет функцию `status()` для отображения сообщений в строке состояния, функцию `save()` – для сохранения текущей версии заметки на сервере и функцию `sync()` – для синхронизации серверной и локальной копии заметки. Функции `save()` и `sync()` используют приемы управления протоколом HTTP, описанные в главе 18. (Интересно отметить, что функция `save()` использует HTTP-метод «PUT» вместо более типичного для таких случаев метода «POST».)

Помимо этих трех основных функций в примере 20.7 определяются также обработчики событий. Чтобы обеспечить синхронизацию локальной и серверной копий заметки, приложению требуется довольно много обработчиков событий:

`onload`

Пытается загрузить заметку с сервера, если там хранится более новая ее версия, и по завершении синхронизации разрешает доступ к окну редактора. Функции `save()` и `sync()` выполняют HTTP-запросы и регистрируют обработчик события «onload» в объекте `XMLHttpRequest`, чтобы определить момент, когда выгрузка или загрузка будут завершены.

`onbeforeunload`

Сохраняет текущую версию заметки на сервере, если она еще не была выгружена.

`oninput`

Всякий раз, когда текст в элементе `<textarea>` изменяется, он сохраняется в объекте `localStorage`, и запускается таймер. Если пользователь не продолжит редактирование в течение 5 секунд, заметка будет выгружена на сервер.

`onoffline`

Когда броузер переключается в автономный режим, в строке состояния выводится сообщение.

`ononline`

Когда броузер подключается к сети, выполняется проверка наличия на сервере более новой версии заметки и выполняется сохранение текущей версии.

`onupdateready`

Если появилась новая версия приложения, выводится сообщение в строке состояния, сообщающее об этом пользователю.

`onnoupdate`

Если приложение не изменилось, сообщает пользователю, что он или она работает с текущей версией.

А теперь, после краткого обзора логики работы приложения PermaNote, в примере 20.7 приводится ее реализация.

Пример 20.7. permanote.js

```
// Некоторые необходимые переменные
var editor, statusline, savebutton, idletimer;

// При первой загрузке приложения
window.onload = function() {
    // Инициализировать локальное хранилище, если это первый запуск
    if (localStorage.note == null) localStorage.note = "";
    if (localStorage.lastModified == null) localStorage.lastModified = 0;
    if (localStorage.lastSaved == null) localStorage.lastSaved = 0;

    // Отыскать элементы, которые составляют пользовательский интерфейс редактора.
    // Инициализировать глобальные переменные.
    editor = document.getElementById("editor");
    statusline = document.getElementById("statusline");
    savebutton = document.getElementById("savebutton");

    editor.value = localStorage.note; // Восстановить сохраненную заметку
    editor.disabled = true;           // Но запретить редактирование до синхр.

    // При вводе нового текста в элемент textarea
    editor.addEventListener("input",
                            function (e) {
                                // Сохранить новую заметку в localStorage
                                localStorage.note = editor.value;
                                localStorage.lastModified = Date.now();
                                // Переустановить таймер ожидания
```

```
                                    if (idletimer) clearTimeout(idletimer);
                                    idletimer = setTimeout(save, 5000);
                                    // Разрешить кнопку сохранения
                                    savebutton.disabled = false;
                                },
                                false);

    // При каждой загрузке приложения пытаться синхронизироваться с сервером
    sync();
};

// Сохраняет заметку на сервере перед уходом со страницы
window.onbeforeunload = function() {
    if (localStorage.lastModified > localStorage.lastSaved)
        save();
};

// Сообщить пользователю перед переходом в автономный режим
window.onoffline = function() { status("Автономный режим"); }

// При подключении к сети выполнить синхронизацию.
window.ononline = function() { sync(); };

// Сообщить пользователю, если доступна новая версия приложения.
// Здесь можно было бы выполнить перезагрузку принудительно, вызвав
// метод location.reload()
window.applicationCache.onupdateready = function() {
    status("Доступна новая версия приложения. " +
           "Чтобы использовать ее, необходимо перезагрузить приложение ");
};

// Также сообщить пользователю, если он использует последнюю версию приложения.
window.applicationCache.onnoupdate = function() {
    status("Вы используете последнюю версию приложения.");
};

// Функция отображения сообщения в строке состояния
function status(msg) { statusline.innerHTML = msg; }

// Выгружает текст заметки на сервер (если сеть подключена).
// Автоматически вызывается через 5 секунд простоя после изменения текста заметки.
function save() {
    if (idletimer) clearTimeout(idletimer);
    idletimer = null;

    if (navigator.onLine) {
        var xhr = new XMLHttpRequest();
        xhr.open("PUT", "/note");
        xhr.send(editor.value);
        xhr.onload = function() {
            localStorage.lastSaved = Date.now();
            savebutton.disabled = true;
        };
    }
}

// Проверяет наличие новой версии заметки на сервере. Если она отсутствует,
// сохраняет текущую версию на сервере.
```

```
function sync() {
    if (navigator.onLine) {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", "/note");
        xhr.send();
        xhr.onload = function() {
            var remoteModTime = 0;
            if (xhr.status == 200) {
                var remoteModTime = xhr.getResponseHeader("Last-Modified");
                remoteModTime = new Date(remoteModTime).getTime();
            }

            if (remoteModTime > localStorage.lastModified) {
                status("На сервере найдена более новая заметка.");
                var useit =
                    confirm("На сервере имеется более новая версия\n" +
                            "заметки. Щелкните на кнопке Ok, чтобы\n" +
                            "использовать эту версию, или на кнопке\n"+
                            "Отмена, чтобы продолжить редактировать\n"+
                            "текущую версию и затереть версию на сервере ");
                var now = Date.now();
                if (useit) {
                    editor.value = localStorage.note = xhr.responseText;
                    localStorage.lastSaved = now;
                    status("Загружена более новая версия.");
                }
                else
                    status("Игнорируется более новая версия заметки.");
                localStorage.lastModified = now;
            }
            else
               status("Редактируется последняя версия заметки.");
            if (localStorage.lastModified > localStorage.lastSaved) {
                save();
            }
            editor.disabled = false; // Разрешить доступ к редактору
            editor.focus();          // И поместить в него курсор ввода
        }
    }
    else { // В автономном режиме мы не можем синхронизироваться
        status("Невозможно синхронизироваться в автономном режиме");
        editor.disabled = false;
        editor.focus();
    }
}
```

21

Работа с графикой и медиафайлами на стороне клиента

В этой главе рассказывается о том, как манипулировать изображениями, управлять аудио- и видеопотоками и рисовать графику. В разделе 21.1 описываются традиционные приемы реализации визуальных эффектов на языке JavaScript, таких как смена изображений, когда одно статическое изображение сменяется другим при наведении указателя мыши. В разделе 21.2 описываются элементы `<audio>` и `<video>`, определяемые стандартом HTML5, и их прикладные интерфейсы в языке JavaScript.

После первых двух разделов, посвященных работе с изображениями, аудио- и видеопотоками, будет рассказано о двух мощных технологиях рисования графических изображений на стороне клиента. Способность воспроизводить на стороне клиента сложные графические изображения имеет важное значение по нескольким причинам:

- Объем программного кода, создающего изображение на стороне клиента, обычно много меньше, чем объем самого изображения, что позволяет сберечь существенную долю полосы пропускания.
- Динамическое создание графических изображений потребляет существенные ресурсы центрального процессора. Переложив эту задачу на клиента, можно существенно снизить нагрузку на сервер и немного сэкономить на стоимости аппаратных средств для него.
- Создание графики на стороне клиента прекрасно согласуется с положениями современной архитектуры веб-приложений, в которой серверы поставляют данные, а клиенты управляют представлением этих данных.

В разделе 21.3 описывается Scalable Vector Graphics, или SVG. SVG – это язык разметки, основанный на языке XML, предназначенный для описания графических изображений. Изображения на языке SVG можно создавать и изменять в сценариях на языке JavaScript с использованием модели DOM. В заключение в разделе 21.4 мы познакомимся с элементом `<canvas>`, определяемым стандартом HTML5, и его обширным прикладным интерфейсом на языке JavaScript, обеспечивающим возможность создания графических изображений. Элемент `<canvas>` является революционной технологией, и он подробно описан в этом разделе.

21.1. Работа с готовыми изображениями

Готовые изображения могут включаться в веб-страницы с помощью тега `<img>`. Подобно любому HTML-элементу, элементом `<img>` можно управлять: присваивание нового URL-адреса свойству `src` заставляет броузер загрузить (если необходимо) и отобразить новое изображение. (Кроме того, можно изменять ширину и высоту изображения, заставляя броузер увеличивать или уменьшать его, но этот прием здесь не рассматривается.)

Возможность динамической замены одного изображения другим в HTML-документе открывает доступ к некоторым специальным эффектам. На практике чаще всего прием смены изображений привязывается к наведению указателя мыши на изображение. Когда изображение размещается внутри гиперссылки, смена изображения становится приглашением пользователю щелкнуть на изображении. (Аналогичный эффект можно получить с помощью псевдокласса CSS `:hover`, позволяющего изменять фоновое изображение элемента.) Следующий фрагмент разметки HTML выводит изображение, которое изменяется при наведении на него указателя мыши:

```
<img src="images/help.gif"
     onmouseover="this.src='images/help_rollover.gif'"
     onmouseout="this.src='images/help.gif'">
```

Обработчики событий в элементе `<img>` изменяют значение свойства `src`, когда указатель мыши наводится на изображение или покидает его границы. Эффект смены изображений отчетливо связывается с возможностью щелкнуть на изображении, поэтому такие элементы `<img>` следует заключать в элементы `<a>` или передавать обработчику события `onclick`.

Чтобы радовать глаз, эффект смены изображений (и родственные ему эффекты) должен иметь минимальное время отклика. Это означает, что необходим некоторый способ, гарантирующий предварительную загрузку всех необходимых изображений в кэш броузера. Для этой цели в клиентском JavaScript имеется специальный прикладной интерфейс: чтобы принудительно поместить изображение в кэш, нужно сначала создать объект `Image` с помощью конструктора `Image()`. Затем, записав в свойство `src` требуемый URL-адрес, загрузить изображение. Это изображение не будет добавлено в документ, поэтому, хотя изображение будет невидимо, броузер загрузит его и поместит в свой кэш. Позднее, когда тот же URL-адрес будет использоваться для изменения изображения, находящегося на экране, изображение быстро загрузится из кэша броузера.

Приведенный выше фрагмент разметки, воспроизводящий эффект смены изображений, не выполняет предварительную загрузку изображений, поэтому пользователь может заметить задержку при смене изображения, когда первый раз наведет на него указатель мыши. Чтобы исправить ситуацию, необходимо немного изменить разметку:

```
<script>(new Image()).src = "images/help_rollover.gif";</script>
<img src="images/help.gif"
```

```
onmouseover="this.src='images/help_rollover.gif'"
onmouseout="this.src='images/help.gif'">
```

21.1.1. Ненавязчивая реализация смены изображений

Только что продемонстрированный фрагмент содержит один элемент `<script>` и два атрибута обработчиков событий с JavaScript-кодом для реализации единственного эффекта смены изображений. Это прекрасный пример *навязчивого* JavaScript-кода: объем программного кода достаточно велик, чтобы осложнить чтение разметки HTML. В примере 21.1 приводится ненавязчивая альтернатива, позволяющая выполнять смену изображений простым добавлением атрибута `data-rollover` (раздел 15.4.3) к любому элементу `<img>`. Обратите внимание, что в этом примере используется функция `onLoad()` из примера 13.5. В нем также используется массив `document.images[]` (раздел 15.2.3), в котором хранятся ссылки на все элементы `<img>` в документе.

Пример 21.1. Ненавязчивая реализация эффекта смены изображений

```
/**
 * rollover.js: Ненавязчивая реализация эффекта смены изображений.
 *
 * Для создания эффекта смены изображений подключите этот модуль к своему HTML-файлу
 * и используйте атрибут data-rollover в элементах <img>, чтобы определить URL-адрес
 * сменного изображения. Например:
 *
 * <img src="normal_image.png" data-rollover="rollover_image.png">
 *
 * Обратите внимание, что для работы этого модуля необходимо подключить onLoad.js
 */
onLoad(function() { // Все в одной анонимной функции: не определяет имен
    // Цикл по всем изображениям, отыскивает атрибут data-rollover
    for(var i = 0; i < document.images.length; i++) {
        var img = document.images[i];
        var rollover = img.getAttribute("data-rollover");
        if (!rollover) continue; // Пропустить изображения без data-rollover

        // Обеспечить загрузку сменного изображения в кэш
        (new Image()).src = rollover;

        // Определить атрибут для сохранения URL-адреса
        // изображения по умолчанию
        img.setAttribute("data-rollout", img.src);

        // Зарегистрировать обработчики событий,
        // создающие эффект смены изображений
        img.onmouseover = function() {
            this.src = this.getAttribute("data-rollover");
        };
        img.onmouseout = function() {
            this.src = this.getAttribute("data-rollout");
        };
    }
});
```

21.2. Работа с аудио- и видеопотоками

Стандарт HTML5 определяет новые элементы `<audio>` и `<video>`, которые теоретически так же просты в использовании, как элемент `<img>`. В броузерах с поддержкой стандарта HTML5 больше не нужно использовать дополнительные расширения (такие как Flash), чтобы внедрить в свои HTML-документы аудио- и видеоклипы:

```
<audio src="background_music.mp3"/>
<video src="news.mov" width=320 height=240/>
```

Однако на практике работать с этими элементами несколько сложнее, чем было показано выше. Производители броузеров не смогли прийти к соглашению о стандарте на аудио- и видеокодеки, которые поддерживались бы всеми броузерами, вследствие чего обычно приходится использовать элементы `<source>`, чтобы указать несколько источников мультимедийных данных в различных форматах:

```
<audio id="music">
<source src="music.mp3" type="audio/mpeg">
<source src="music.ogg" type='audio/ogg; codec="vorbis"'>
</audio>
```

Обратите внимание, что элементы `<source>` не имеют содержимого: они не имеют закрывающего тега `</source>`, и от вас не требуется завершать их последовательностью символов `/>`.

Броузеры, поддерживающие элементы `<audio>` и `<video>`, не будут отображать их содержимое. Тогда как броузеры, не поддерживающие их, отобразят это содержимое. Чтобы решить эту проблему, можно вставить внутрь содержимое для обратной совместимости (например, элемент `<object>`, который вызывает расширение Flash):

```
<video id="news" width=640 height=480 controls preload>
  <!-- В формате WebM для Firefox и Chrome -->
  <source src="news.webm" type='video/webm; codecs="vp8, vorbis"'>
  <!-- В формате H.264 для IE и Safari -->
  <source src="news.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
  <!-- Для совместимости с расширением Flash -->
  <object width=640 height=480 type="application/x-shockwave-flash"
          data="flash_movie_player.swf">
    <!-- Здесь можно указать параметры настройки проигрывателя Flash -->
    <!-- Текстовое содержимое, используемое в самом худшем случае -->
    <div>Элемент video не поддерживается и расширение Flash не установлено.</div>
  </object>
</video>
```

Элементы `<audio>` и `<video>` поддерживают атрибут `controls`. Если он присутствует (или соответствующее JavaScript-свойство имеет значение `true`), они будут отображать элементы управления, включая кнопки запуска на воспроизведение и паузы, регулятор громкости и т. д. Но кроме этого, элементы `<audio>` и `<video>` предоставляют прикладной интерфейс, обеспечивающий широкие возможности управления воспроизведением, с помощью которого вы можете добавлять простые звуковые эффекты в свои веб-приложения или создавать собственные панели управления воспроизведением. Несмотря на различия во внешнем виде, элементы

`<audio>` и `<video>` предоставляют практически один и тот же прикладной интерфейс (единственное отличие между которыми состоит в том, что элемент `<video>` имеет свойства `width` и `height`), поэтому большая часть того, что рассказывается далее в этом разделе, в равной степени относится к обоим элементам.

Несмотря на раздражающую необходимость определять мультимедийные данные в нескольких форматах, возможность воспроизводить звук и видеоизображение родными средствами броузера без использования дополнительных расширений является новой мощной особенностью, добавленной стандартом HTML5. Обратите внимание, что обсуждение проблемы поддержки кодеков и совместимости броузеров выходит далеко за рамки этой книги. В следующих подразделах мы сосредоточимся исключительно на методах JavaScript, предназначенных для работы с аудио- и видеопотоками.

Конструктор Audio()

Элементы `<audio>` не имеют визуального представления в документе, если не установить атрибут `controls`. И так же, как имеется возможность создавать неотображаемые изображения с помощью конструктора `Image()`, механизм поддержки мультимедиа, определяемый стандартом HTML5, позволяет создавать аудиоэлементы с помощью конструктора `Audio()`, передавая ему аргумент с URL-адресом источника данных:

```
new Audio("chime.wav").play(); // Загрузить и проиграть звуковой эффект
```

Конструктор `Audio()` возвращает тот же объект, который будет получен при обращении к элементу `<audio>` в документе или при создании нового аудиоэлемента вызовом `document.createElement("audio")`. Обратите внимание, что все вышесказанное относится только к аудиоэлементам: механизм поддержки мультимедиа не имеет соответствующего конструктора `Video()`.

21.2.1. Выбор типа и загрузка

Если вам потребуется проверить, способен ли мультимедийный элемент воспроизводить мультмедийные данные в определенном формате, передайте MIME-тип этих данных (при необходимости с параметром `codec`) методу `canPlayType()`. Элемент вернет пустую строку (ложное значение), если он не способен проигрывать мультимедийные данные в этом формате. В противном случае он вернет строку «maybe» (возможно) или «probably» (вероятно). Из-за сложной природы аудио- и видеокодеков проигрыватель в общем случае не может сообщить ничего более определенного, чем «probably» (вероятно), не предприняв фактическую попытку загрузить и воспроизвести данные указанного типа:

```
var a = new Audio();
if (a.canPlayType("audio/wav")) {
    a.src = "soundeffect.wav";
    a.play();
}
```

Когда свойству `src` мультимедийного элемента присваивается значение, он начинает процесс загрузки мультимедийных данных. (Этот процесс не продвинется слишком далеко, если не установить в свойстве `preload` значение «auto».) Присваивание нового значения свойству `src` во время загрузки или воспроизведения других мультимедийных данных прервет загрузку или воспроизведение старых данных. Если вместо настройки атрибута `src` вы будете добавлять в мультимедийный элемент элементы `<source>`, то он не сможет приступить к выбору нужного элемента, так как не будет знать, когда закончится формирование полного комплекта элементов `<source>`, и не сможет начать загрузку данных, пока явно не будет вызван метод `load()`.

21.2.2. Управление воспроизведением

Самыми важными методами элементов `<audio>` и `<video>` являются методы `play()` и `pause()`, которые запускают и останавливают воспроизведение:

```
// Когда документ будет загружен, запустить фоновое проигрывание мелодии
window.addEventListener("load", function() {
                            document.getElementById("music").play();
                        }, false);
```

Помимо возможности запустить и остановить проигрывание звука или видео имеется возможность выполнить переход к требуемому месту в мультимедийных данных установкой свойства `currentTime`. Это свойство определяет время в секундах, к которому должен быть выполнен переход, и его можно устанавливать в процессе проигрывания данных или во время паузы. (Свойства `initialTime` и `duration` ограничивают диапазон допустимых значений свойства `currentTime`; подробнее об этих свойствах рассказывается ниже.)

Свойство `volume` определяет уровень громкости как числовое значение в диапазоне от 0 (минимальная громкость) до 1 (максимальная громкость). Свойству `muted` может быть присвоено значение `true`, чтобы выключить звук, или `false`, чтобы продолжить воспроизведение с установленным уровнем громкости.

Свойство `playbackRate` определяет скорость проигрывания. Значение 1,0 соответствует нормальной скорости. Значения выше 1 соответствуют «ускоренному воспроизведению вперед», а значения от 0 до 1 – «замедленному воспроизведению вперед». Отрицательные значения предполагают проигрывание звука или видео в обратном направлении, но на момент написания этих строк броузеры не поддерживали такую возможность. Элементы `<audio>` и `<video>` также имеют свойство `defaultPlaybackRate`. Всякий раз, когда вызывается метод `play()`, значение свойства `defaultPlaybackRate` присваивается свойству `playbackRate`.

Обратите внимание, что свойства `currentTime`, `volume`, `muted` и `playbackRate` не являются единственными средствами управления воспроизведением. Если элемент `<audio>` или `<video>` имеет атрибут `controls`, он отображает элементы управления проигрывателем, давая пользователю возможность управлять воспроизведением. В этом случае сценарий может читать значения таких свойств, как `muted` и `currentTime`, чтобы определить, как протекает воспроизведение мультимедийных данных.

HTML-атрибуты `controls`, `loop`, `preload` и `autoplay` оказывают влияние на воспроизведение аудио и видео, а также доступны для чтения и записи как JavaScript-свойства. Атрибут `controls` определяет, должны ли отображаться элементы управления проигрывателем. Присвойте этому свойству значение `true`, чтобы отобразить элементы управления, или `false`, чтобы скрыть их. Логическое свойство `loop` определяет, должно ли воспроизведение начинаться сначала по достижении конца (`true`) или нет (`false`). Свойство `preload` определяет, какой объем мультимедийных данных должен быть загружен прежде, чем пользователь сможет запустить проигрывание. Значение «none» означает, что предварительная загрузка данных не требуется. Значение «metadata» означает, что предварительно должны быть загружены такие метаданные, как продолжительность, битрейт и размер кадра, но предварительная загрузка самих данных не требуется. При отсутствии атрибута `preload` броузеры обычно загружают только метаданные. Значение «auto» означает, что броузер должен предварительно загрузить такой объем данных, какой он сочтет нужным. Наконец, свойство `autoplay` определяет, должно ли начаться воспроизведение автоматически, после загрузки достаточного объема данных. Присваивание свойству `autoplay` значения `true` подразумевает, что броузер должен предварительно загрузить некоторый объем данных.

21.2.3. Определение состояния мультимедийных элементов

Элементы `<audio>` и `<video>` имеют несколько свойств, доступных только для чтения, которые описывают текущее состояние данных и проигрывателя. Свойство `paused` имеет значение `true`, если проигрывание было приостановлено. Свойство `seeking` имеет значение `true`, если проигрыватель выполняет переход к новой позиции в проигрываемых данных. Свойство `ended` имеет значение `true`, если проигрыватель достиг конца и остановился. (Свойство `ended` никогда не приобретет значение `true`, если свойству `loop` было присвоено значение `true`.)

Свойство `duration` определяет продолжительность проигрываемых данных в секундах. Если прочитать это свойство до того, как будут получены метаданные, оно вернет значение `NaN`. Для потоковых данных с неопределенной продолжительностью, например, при прослушивании Интернет-радио, это свойство возвращает значение `Infinity`.

Свойство `initialTime` определяет начальное время в проигрываемых данных в секундах. Для мультимедийных клипов с фиксированной продолжительностью это свойство обычно имеет значение 0. Для потоковых данных это свойство возвращает самое раннее время данных в буфере, к которому еще можно вернуться. Свойство `currentTime` не может быть установлено в значение меньше чем значение свойства `initialTime`.

Три других свойства позволяют получить более точное представление о временной шкале для проигрываемых данных и состоянии механизма буферизации. Свойство `played` возвращает диапазон или диапазоны времени, проигрываемые в настоящее время. Свойство `buffered` возвращает диапазон или диапазоны времени, которые в настоящее время находятся в буфере, а свойство `seekable` возвращает диапазон или диапазоны времени, куда проигрыватель может выполнить переход. (Эти свойства можно использовать для реализации индикатора, иллюстрирующего свойства `currentTime` и `duration`, а также продолжительность воспроизведенных данных и объем данных в буфере.)

Свойства `played`, `buffered` и `seekable` являются объектами `TimeRanges`. Каждый объект имеет свойство `length`, определяющее количество представляемых им диапазонов, и методы `start()` и `end()`, возвращающие начало и конец (в секундах) диапазона с указанным номером. В наиболее типичном случае, когда имеется всего один непрерывный диапазон, эти методы вызываются, как `start(0)` и `end(0)`. Если, к примеру, предположить, что переходы не выполнялись и данные буферизованы с самого начала, то можно использовать следующий прием, чтобы определить, какая доля ресурса в процентах была загружена в буфер:

```
var percent_loaded = Math.floor(song.buffered.end(0) / song.duration * 100);
```

Наконец, имеются еще три свойства, `readyState`, `networkState` и `error`, позволяющие получить низкоуровневую информацию о состоянии элементов `<audio>` и `<video>`. Все эти свойства имеют числовые значения, и для каждого допустимого значения определена константа. Обратите внимание, что эти константы определены непосредственно в мультимедийном объекте (или в объекте ошибки). Эти константы можно использовать, как показано ниже:

```
if (song.readyState === song.HAVE_ENOUGH_DATA) song.play();
```

Свойство `readyState` определяет, как много мультимедийных данных было загружено, и, соответственно, готов ли элемент начать воспроизведение этих данных. Допустимые значения этих свойств и их смысл перечислены ниже:

Константа	Значение	Описание
`HAVE_NOTHING`	0	Мультимедийные данные и метаданные еще не были загружены.
`HAVE_METADATA`	1	Метаданные были загружены, но мультимедийные данные для текущей позиции воспроизведения еще не были загружены. Это означает, что имеется возможность узнать продолжительность воспроизведения или размеры кадра видеоклипа, а также выполнить переход к другой позиции, установив свойство `currentTime`, но броузер в настоящий момент не может воспроизводить в позиции `currentTime`.
`HAVE_CURRENT_DATA`	2	Мультимедийные данные для позиции `currentTime` были загружены, но объем загруженных данных еще недостаточен, чтобы начать воспроизведение. Для видеоклипов это обычно означает, что текущий кадр был загружен, а следующий еще нет. Это состояние возникает в конце аудио- или видеоклипа.
`HAVE_FUTURE_DATA`	3	Был загружен достаточный объем данных, чтобы начать воспроизведение, но их недостаточно, чтобы воспроизвести клип до конца без приостановки на загрузку дополнительных данных.
`HAVE_ENOUGH_DATA`	4	Был загружен достаточный объем данных, чтобы их, скорее всего, можно было воспроизвести до конца, без приостановки.

Свойство `networkState` определяет, использует ли (и если нет, то почему) сеть мультимедийный элемент:

Константа	Значение	Описание
`NETWORK_EMPTY`	0	Элемент еще не приступил к использованию сети. Это состояние возникает, например, перед установкой атрибута `src`.

Константа	Значение	Описание
`NETWORK_IDLE`	1	В настоящее время элемент не загружает данные из сети. Возможно, он загрузил ресурс полностью или сохранил в буфере достаточный объем требуемых данных. Или, возможно, свойству `preload` было присвоено значение «none» и элементу еще не была дана команда загружать или воспроизводить клип.
`NETWORK_LOADING`	2	В настоящее время элемент загружает данные из сети.
`NETWORK_NO_SOURCE`	3	Элемент не может отыскать источник данных, которые требуется воспроизвести.

Когда при загрузке или воспроизведении возникает ошибка, броузер записывает определенное значение в свойство `error` элемента `<audio>` или `<video>`. При отсутствии ошибок свойство `error` имеет значение `null`. Иначе оно ссылается на объект с числовым свойством `code`, описывающим ошибку. Объект ошибки также определяет константы для возможных кодов ошибок:

Константа	Значение	Описание
`MEDIA_ERR_ABORTED`	1	Пользователь потребовал остановить загрузку клипа.
`MEDIA_ERR_NETWORK`	2	Мультимедийные данные имеют верный тип, но их загрузке препятствуют ошибки в сети.
`MEDIA_ERR_DECODE`	3	Мультимедийные данные имеют верный тип, но их декодированию и воспроизведению препятствуют ошибки кодирования.
`MEDIA_ERR_SRC_NOT_SUPPORTED`	4	Мультимедийные данные, на которые ссылается атрибут `src`, имеют тип, который не может воспроизводиться броузером.

Свойство `error` можно использовать, как показано ниже:

```
if (song.error.code == song.error.MEDIA_ERR_DECODE)
    alert("Невозможно воспроизвести песню: повреждены аудиоданные.");
```

21.2.4 События мультимедийных элементов

Элементы `<audio>` и `<video>` являются довольно сложными элементами – они должны откликаться на взаимодействие пользователя с их элементами управления, выполнять сетевые операции и даже, в процессе воспроизведения, реагировать на простое течение времени. Мы только что видели, что эти элементы имеют довольно много свойств, определяющих их состояние. Подобно большинству HTML-элементов, элементы `<audio>` и `<video>` генерируют события при изменении своего состояния. Поскольку состояние этих элементов описывается множеством характеристик, они могут генерировать довольно много различных событий.

В таблице ниже перечислены 22 события мультимедийных элементов, примерно в том порядке, в каком они обычно возникают. Элементы не имеют свойств обработчиков этих событий, поэтому для регистрации функций обработчиков в элементах `<audio>` и `<video>` следует использовать метод `addEventListener()`.

Тип события	**Описание**
`loadstart`	Возбуждается, когда элемент отправляет запрос на получение мультимедийных данных.Свойство `networkState` имеет значение `NETWORK_LOADING`.
`progress`	Загрузка мультимедийных данных продолжается. Свойство `networkState` имеет значение `NETWORK_LOADING`. Это событие обычно возбуждается от 2 до 8 раз в секунду.
`loadedmetadata`	Метаданные загружены, и доступна информация о продолжительности клипа и размерах кадра. Свойство `readyState` в первый раз получает значение `HAVE_METADATA`.
`loadeddata`	Данные для текущей позиции воспроизведения загружены, а свойство `readyState` получает значение `HAVE_CURRENT_DATA`.
`canplay`	Загружен достаточный объем данных, чтобы начать воспроизведение, но, скорее всего, необходимо загрузить в буфер дополнительный объем данных. Свойство `readyState` имеет значение `HAVE_FUTURE_DATA`.
`canplaythrough`	Загружен достаточный объем данных, чтобы наверняка воспроизвести клип до конца без дополнительных пауз для буферизации дополнительных данных. Свойство `readyState` имеет значение `HAVE_ENOUGH_DATA`.
`suspend`	Элемент загрузил в буфер достаточный объем данных и временно приостановил загрузку. Свойство `networkState` получает значение `NETWORK_IDLE`.
`stalled`	Элемент пытается загрузить данные, но данные не поступают из сети. Свойство `networkState` остается в состоянии `NETWORK_LOADING`.
`play`	Был вызван метод `play()` или воспроизведение было автоматически запущено из-за наличия атрибута `autoplay`. Если был загружен достаточный объем данных, вслед за этим событием последует событие «playing». Иначе последует событие «waiting».
`waiting`	Воспроизведение не может быть начато или оно было приостановлено из-за недостатка данных в буфере. Когда будет загружен достаточный объем данных, последует событие «playing».
`playing`	Начато воспроизведение клипа.
`timeupdate`	Изменилось значение свойства `currentTime`. В процессе воспроизведения это событие генерируется от 4 до 60 раз в секунду в зависимости от нагрузки на систему и скорости выполнения обработчиков событий.
`pause`	Был вызван метод `pause()`, и воспроизведение было приостановлено.
`seeking`	Сценарий или пользователь потребовал перейти к участку клипа, отсутствующему в буфере, и воспроизведение было остановлено до загрузки данных. Свойство `seeking` получает значение `true`.
`seeked`	Свойство `seeking` получает значение `false`.
`ended`	Воспроизведение было остановлено по достижении конца клипа.
`durationchange`	Изменилось значение свойства `duration`.
`volumechange`	Изменилось значение свойства `volume` или `muted`.
`ratechange`	Изменилось значение свойства `playbackRate` или `defaultPlaybackRate`.
`abort`	Элемент прекратил загрузку данных. Обычно это происходит по требованию пользователя. Свойство `error.code` получает значение `MEDIA_ERR_ABORTED`.
`error`	Сетевая или какая-либо другая ошибка не дает возможности загрузить данные. Свойство `error.code` получает любое другое значение, отличное от `MEDIA_ERR_ABORTED`.
`emptied`	В результате события «error» или «abort» свойство `networkState` получило значение `NETWORK_EMPTY`.

21.3. SVG – масштабируемая векторная графика

Масштабируемая векторная графика (SVG) – это грамматика языка XML для описания графических изображений. Слово «векторная» в названии указывает на фундаментальное отличие от таких форматов растровой графики, как GIF, JPEG и PNG, где изображение задается матрицей пикселов. Формат SVG представляет собой точное, не зависящее от разрешения (отсюда слово «масштабируемая») описание шагов, которые необходимо выполнить, чтобы нарисовать требуемый рисунок. Вот пример простого SVG-изображения в текстовом формате:

```
<!-- Начало рисунка и объявление пространства имен -->
<svg xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 1000 1000">  <!-- Система координат рисунка -->
  <defs>                       <!-- Настройка некоторых определений -->
    <linearGradient id="fade"> <!-- Цветовой градиент с именем "fade" -->
      <stop offset="0%" stop-color="#008"/> <!-- Начинаем с темно-синего -->
      <stop offset="100%" stop-color="#ccf"/><!--Заканчиваем светло-синим-->
    </linearGradient>
  </defs>
<!--
    Нарисовать прямоугольник с тонкой черной рамкой и заполнить его градиентом
-->
  <rect x="100" y="200" width="800" height="600"
    stroke="black" stroke-width="25" fill="url(#fade)"/>
</svg>
```

На рис. 21.1 показано графическое представление этого SVG-файла.

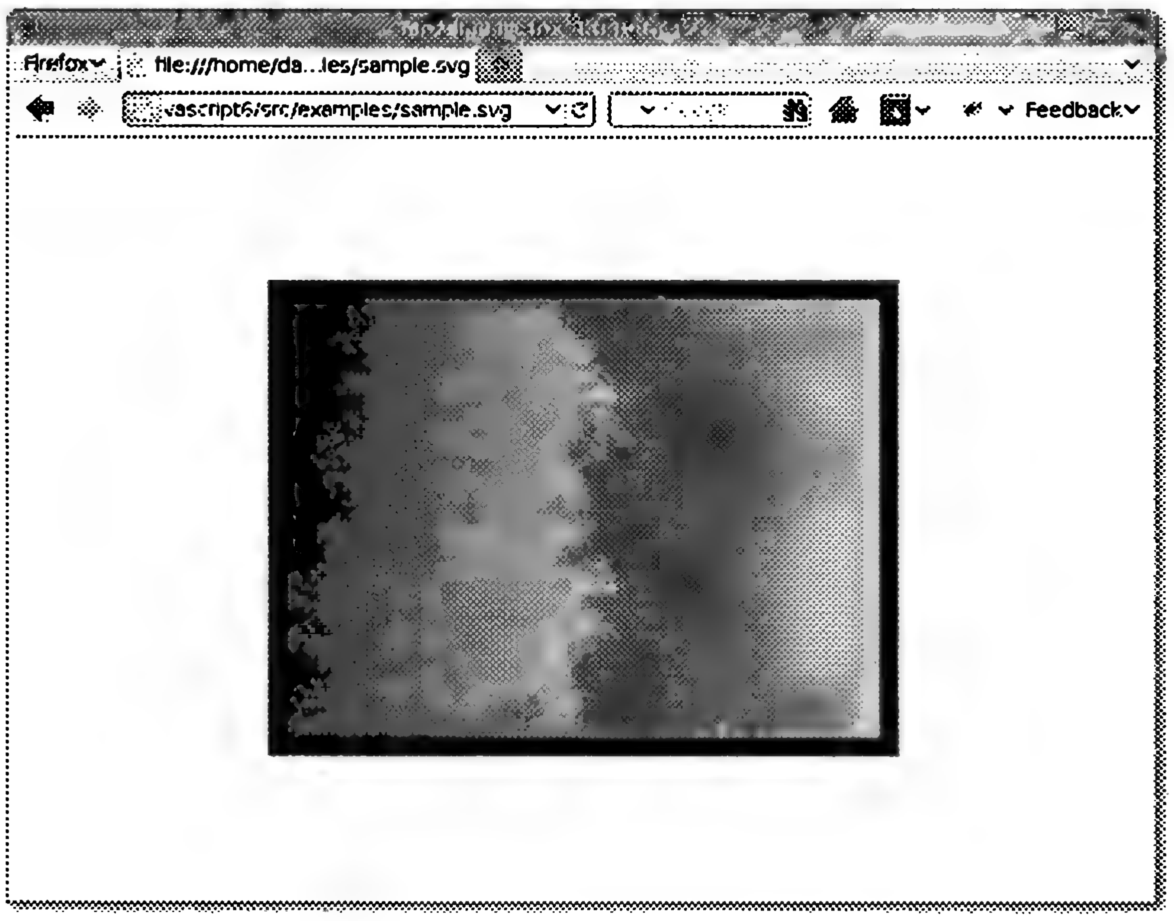

Firefox
file:///home/da...les/sample.svg
vascript6/src/examples/sample.svg
Feedback

Рис. 21.1. Простое изображение в формате SVG

SVG – это довольно обширная грамматика умеренной сложности. Помимо простых примитивов рисования она позволяет воспроизводить произвольные кривые, текст и анимацию. Рисунки в формате SVG могут даже содержать JavaScript-сценарии и таблицы CSS-стилей, что позволяет наделить их информацией о поведении и представлении. В этом разделе показано, как с помощью клиентского JavaScript-кода (встроенного в HTML-, а не в SVG-документ) можно динамически создавать графические изображения средствами SVG. Приводимые здесь примеры SVG-изображений позволяют лишь отчасти оценить возможности формата SVG. Полное описание этого формата доступно в виде обширной, но вполне понятной спецификации, которая поддерживается консорциумом W3C и находится по адресу *http://www.w3.org/TR/SVG/*. Обратите внимание: эта спецификация включает в себя полное описание объектной модели документа (DOM) для SVG-документов. В данном разделе рассматриваются приемы манипулирования SVG-графикой с помощью стандартной модели XML DOM, а модель SVG DOM не затрагивается.

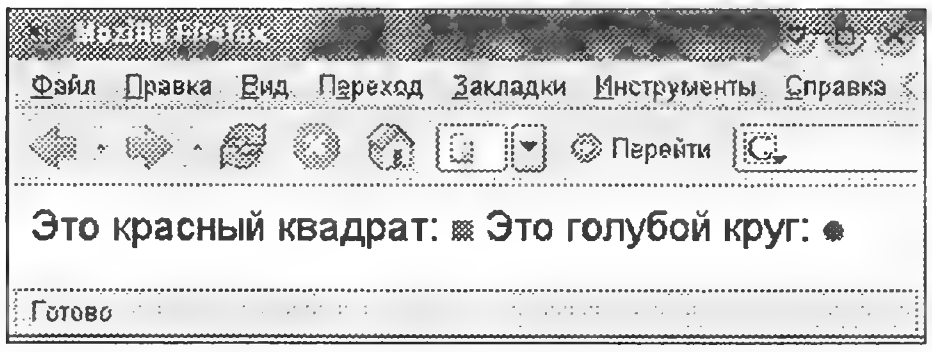
К моменту написания этих строк все текущие веб-броузеры, кроме IE, имели встроенную поддержку формата SVG (она также будет включена в IE9). В последних версиях броузеров отображать SVG-изображения можно с помощью обычного элемента `<img>`. Некоторые немного устаревшие броузеры (такие как Firefox 3.6) не поддерживают такую возможность и требуют использовать для этих целей элемент `<object>`:

```
<object data="sample.svg" type="image/svg+xml" width="100" height="100"/>
```

При использовании в элементе `<img>` или `<object>` SVG можно рассматривать как еще один формат представления графических изображений, который, с точки зрения программиста на языке JavaScript, ничем особенным не выделяется. Гораздо больший интерес представляет сама возможность встраивания SVG-изображений непосредственно в документы и выполнения операций над ними. Поскольку формат SVG является грамматикой языка XML, изображения в этом формате можно встраивать непосредственно в XHTML-документы, как показано ниже:

```
<?xml version="1.0"?>
<!-- Объявить HTML как пространство имен по умолчанию, а SVG - с префиксом "svg:" -->
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:svg="http://www.w3.org/2000/svg">
<body>
Это красный квадрат: <svg:svg width="10" height="10">
  <svg:rect x="0" y="0" width="10" height="10" fill="red"/>
</svg:svg>
Это голубой круг: <svg:svg width="10" height="10">
  <svg:circle cx="5" cy="5" r="5" fill="blue"/>
</svg:svg>
</body>
</html>
```

Этот прием можно использовать во всех текущих броузерах, кроме IE. На рис. 21.2 показано, как Firefox отображает этот XHTML-документ.



Это красный квадрат: ■ Это голубой круг: ●

Рис. 21.2. SVG-графика в XHTML-документе

Стандарт HTML5 сокращает количество различий между XML и HTML и позволяет вставлять разметку на языке SVG (и MathML) непосредственно в HTML-файлы, без объявления пространств имен или префиксов тегов:

```
<!DOCTYPE html>
<html>
<body>
Это красный квадрат: <svg width="10" height="10">
  <rect x="0" y="0" width="10" height="10" fill="red"/>
</svg>
Это голубой круг: <svg width="10" height="10">
  <circle cx="5" cy="5" r="5" fill="blue"/>
</svg>
</body>
</html>
```

На момент написания этих строк непосредственное встраивание SVG-изображений в разметку HTML поддерживали только самые последние версии броузеров.

Так как формат SVG – это грамматика языка XML, рисование SVG-изображений заключается просто в использовании модели DOM для создания соответствующих XML-элементов. В примере 21.2 приводится реализация функции `pieChart()`, которая создает SVG-элементы для воспроизведения круговой диаграммы, подобной той, что показана на рис. 21.3.

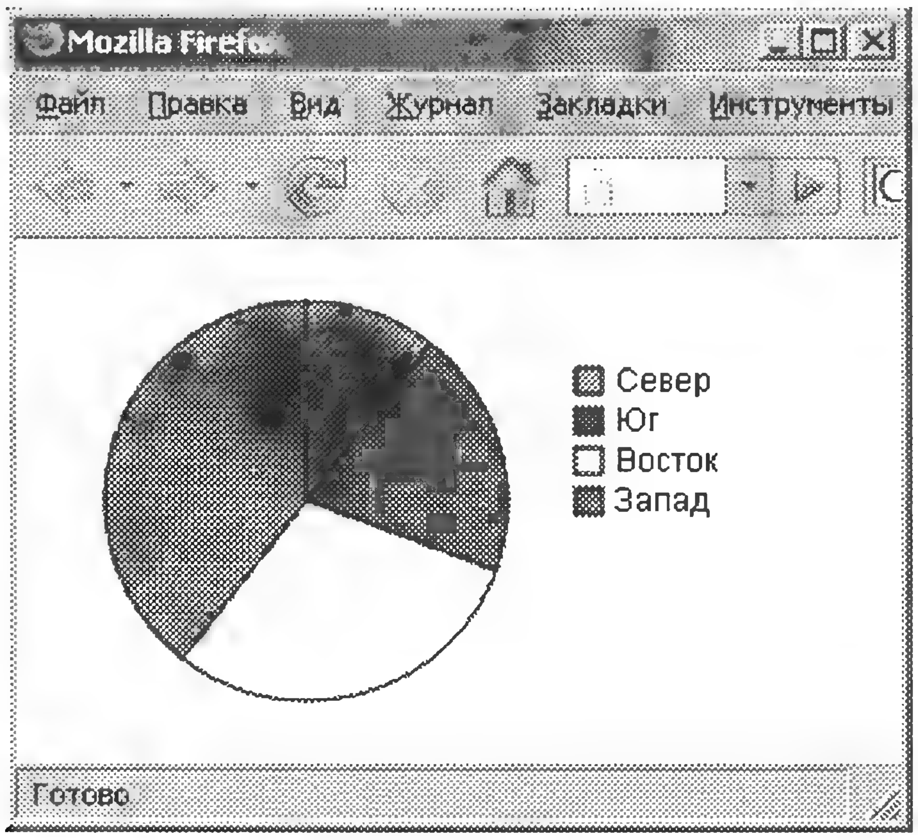


Рис. 21.3. Круговая диаграмма в формате SVG, построенная JavaScript-сценарием

Пример 21.2. Рисование круговой диаграммы средствами JavaScript и SVG

```
/**
 * Создает элемент <svg> и рисует в нем круговую диаграмму.
 * Аргументы:
 *   data: массив чисел для диаграммы, по одному для каждого сектора.
 *   width,height: размеры SVG-изображения в пикселах
 *   cx, cy, r: координаты центра и радиус круга
 *   colors: массив цветов в формате HTML, по одному для каждого сектора
 *   labels: массив меток для легенды, по одной для каждого сектора
 *   lx, ly: координаты левого верхнего угла легенды диаграммы
 * Возвращает:
 *   Элемент <svg>, хранящий круговую диаграмму.
 *   Вызывающая программа должна вставить возвращаемый элемент в документ.
 */
function pieChart(data, width, height, cx, cy, r, colors, labels, lx, ly) {
    // Пространство имен XML для элементов svg
    var svgns = "http://www.w3.org/2000/svg";

    // Создать элемент <svg>, указать размеры в пикселах и координаты
    var chart = document.createElementNS(svgns, "svg:svg");
    chart.setAttribute("width", width);
    chart.setAttribute("height", height);
    chart.setAttribute("viewBox", "0 0 " + width + " " + height);

    // Сложить вместе все значения, чтобы получить общую сумму всей диаграммы
    var total = 0;
    for(var i = 0; i < data.length; i++) total += data[i];

    // Определить величину каждого сектора. Углы измеряются в радианах.
    var angles = []
    for(var i = 0; i < data.length; i++) angles[i] = data[i]/total*Math.PI*2;

    // Цикл по всем секторам диаграммы.
    startangle = 0;
    for(var i = 0; i < data.length; i++) {
        // Точка, где заканчивается сектор
        var endangle = startangle + angles[i];

        // Вычислить координаты точек пересечения радиусов, образующих сектор,
        // с окружностью. В соответствии с выбранными формулами углу 0 радиан
        // соответствует точка в самой верхней части окружности,
        // а положительные значения откладываются от нее по часовой стрелке.
        var x1 = cx + r * Math.sin(startangle);
        var y1 = cy - r * Math.cos(startangle);
        var x2 = cx + r * Math.sin(endangle);
        var y2 = cy - r * Math.cos(endangle);

        // Это флаг для углов, больших половины окружности.
        // Он необходим SVG-механизму рисования дуг
        var big = 0;
        if (endangle - startangle > Math.PI) big = 1;

        // Мы описываем сектор с помощью элемента <svg:path>.
        // Обратите внимание, что он создается вызовом createElementNS()
        var path = document.createElementNS(svgns, "path");

        // Эта строка хранит информацию о контуре, образующем сектор
        var d = "M " + cx + "," + cy + // Начало в центре окружности
            " L " + x1 + "," + y1 +    // Нарисовать линию к точке (x1,y1)
            " A " + r + "," + r +      // Нарисовать дугу с радиусом r
            " 0 " + big + " 1 " +      // Информация о дуге...
            x2 + "," + y2 +            // Дуга заканчивается в точке (x2,y2)
            " Z";                      // Закончить рисование в точке (cx,cy)

        // Теперь установить атрибуты элемента <svg:path>
        path.setAttribute("d", d);              // Установить описание контура
        path.setAttribute("fill", colors[i]);   // Установить цвет сектора
        path.setAttribute("stroke", "black");   // Рамка сектора - черная
        path.setAttribute("stroke-width", "2"); // 2 единицы толщиной
        chart.appendChild(path);                // Вставить сектор в диаграмму

        // Следующий сектор начинается в точке, где закончился предыдущий
        startangle = endangle;

        // Нарисовать маленький квадрат для идентификации сектора в легенде
        var icon = document.createElementNS(svgns, "rect");
        icon.setAttribute("x", lx);            // Координаты квадрата
        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("width", 20);        // Размер квадрата
        icon.setAttribute("height", 20);
        icon.setAttribute("fill", colors[i]); // Тем же цветом, что и сектор
        icon.setAttribute("stroke", "black"); // Такая же рамка.
        icon.setAttribute("stroke-width", "2");
        chart.appendChild(icon);               // Добавить в диаграмму

        // Добавить метку правее квадрата
        var label = document.createElementNS(svgns, "text");
        label.setAttribute("x", lx + 30);      // Координаты текста
        label.setAttribute("y", ly + 30*i + 18);
        // Стиль текста можно также определить посредством таблицы CSS-стилей
        label.setAttribute("font-family", "sans-serif");
        label.setAttribute("font-size", "16");
        // Добавить текстовый DOM-узел в элемент <svg:text>
        label.appendChild(document.createTextNode(labels[i]));
        chart.appendChild(label);              // Добавить текст в диаграмму
    }
    return chart;
}
```

Программный код в примере 21.2 относительно прост. Здесь выполняются некоторые математические расчеты для преобразования исходных данных в углы секторов круговой диаграммы. Однако основную часть примера составляет программный код, создающий SVG-элементы и выполняющий настройку их атрибутов. Чтобы этот пример мог работать в броузерах, не полностью поддерживающих стандарт HTML5, здесь формат SVG интерпретируется как грамматика XML и вместо метода `createElement()` используются пространство имен SVG и метод `createElementNS()`.

Самая малопонятная часть этого примера – программный код, выполняющий рисование сектора диаграммы. Для отображения каждого сектора используется тег `<svg:path>`. Этот SVG-элемент описывает рисование произвольных фигур, состоящих из линий и кривых. Описание фигуры определяется атрибутом `d` элемента `<svg:path>`. Основу описания составляет компактная грамматика символьных кодов и чисел, определяющих координаты, углы и прочие значения. Например, символ M означает «move to» (переместиться в точку), и вслед за ним должны следовать координаты X и Y точки. Символ L означает «line to» (рисовать линию до точки); он рисует линию от текущей точки до точки с координатами, которые следуют далее. Кроме того, в этом примере используется символьный код A, который рисует дугу (arc). Вслед за этим символом следуют семь чисел, описывающих дугу. Точное описание нас здесь не интересует, но вы можете найти его в спецификации по адресу *http://www.w3.org/TR/SVG/*.

Обратите внимание, что функция `pieChart()` возвращает элемент `<svg>`, содержащий описание круговой диаграммы, но она не вставляет этот элемент в документ. Предполагается, что это будет делать вызывающая программа. Диаграмма, изображенная на рис. 21.3, была создана с помощью следующего файла:

```
<html>
<head>
<script src="PieChart.js"></script>
</head>
<body onload="document.body.appendChild(
                pieChart([12, 23, 34, 45], 640, 400, 200, 200, 150,
                         ['красный','синий','желтый','зеленый'],
                         ['Север','Юг', 'Восток', 'Запад'], 400, 100));
            ">
</body>
</html>
```

В примере 21.3 демонстрируется создание еще одного SVG-изображения: в нем формат SVG используется для отображения аналоговых часов (рис. 21.4). Однако вместо создания дерева SVG-элементов с самого начала, в нем используется статическое SVG-изображение циферблата, встроенное в HTML-страницу. Это статическое изображение включает два SVG-элемента `<line>`, представляющих часовую и минутную стрелки. Обе линии направлены вверх, в результате чего статическое изображение часов показывает время 12:00. Чтобы превратить это изображение в действующие часы, в примере используется сценарий на языке JavaScript, устанавливающий атрибут `transform` каждого элемента `<line>` и поворачивающий их на углы, соответствующие текущему времени.

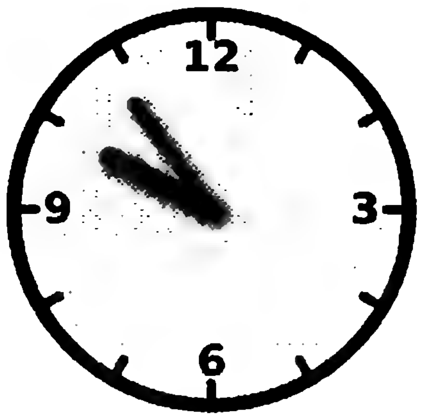


Рис. 21.4. SVG-часы

Обратите внимание, что в примере 21.3 разметка SVG встроена непосредственно в файл HTML5 и в ней не используются пространства имен XML, как в XHTML-файле выше. Это означает, что данный пример будет работать только в броузерах, поддерживающих возможность непосредственного встраивания разметки SVG. Однако если преобразовать HTML-файл в XHTML, тот же самый прием будет работать и в старых броузерах с поддержкой SVG.

Пример 21.3. Отображение времени посредством манипулирования SVG-изображением

```
<!DOCTYPE HTML>
<html>
<head>
<title>Analog Clock</title>
```

```
<script>
function updateTime() {  // Обновляет SVG-изображение часов
                         // в соответствии с текущим временем
    var now = new Date();                      // Текущее время
    var min = now.getMinutes();                // Минуты
    var hour = (now.getHours() % 12) + min/60; // Часы с дробной частью
    var minangle = min*6;                      // 6 градусов на минуту
    var hourangle = hour*30;                   // 30 градусов на час

    // Получить SVG-элементы стрелок часов
    var minhand = document.getElementById("minutehand");
    var hourhand = document.getElementById("hourhand");

    // Установить в них SVG-атрибут для перемещения по циферблату
    minhand.setAttribute("transform", "rotate(" + minangle + ",50,50)");
    hourhand.setAttribute("transform", "rotate(" + hourangle + ",50,50)");

    // Обновлять показания часов 1 раз в минуту
    setTimeout(updateTime, 60000);
}
</script>
<style>
/* Все следующие CSS-стили применяются к SVG-элементам, объявленным ниже */
#clock {                    /* общие стили для всех элементов часов */
    stroke: black;          /* черные линии */
    stroke-linecap: round;  /* с закругленными концами */
    fill: #eef;             /* на светлом, голубовато-сером фоне */
}
#face { stroke-width: 3px;}      /* рамка циферблата часов */
#ticks { stroke-width: 2; }      /* метки часов на циферблате */
#hourhand {stroke-width: 5px;}   /* широкая часовая стрелка */
#minutehand {stroke-width: 3px;} /* узкая минутная стрелка */
#numbers {                       /* стиль отображения цифр на циферблате */
    font-family: sans-serif; font-size: 7pt; font-weight: bold;
    text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body onload="updateTime()">
  <!-- viewBox - система координат, width и height - экранные размеры -->
  <svg id="clock" viewBox="0 0 100 100" width="500" height="500">
    <defs> <!-- Определить фильтр для рисования теней -->
      <filter id="shadow" x="-50%" y="-50%" width="200%" height="200%">
        <feGaussianBlur in="SourceAlpha" stdDeviation="1" result="blur" />
        <feOffset in="blur" dx="1" dy="1" result="shadow" />
        <feMerge>
          <feMergeNode in="SourceGraphic"/><feMergeNode in="shadow"/>
        </feMerge>
      </filter>
    </defs>
    <circle id="face" cx="50" cy="50" r="45"/> <!-- циферблат -->
    <g id="ticks">                             <!-- 12 часовых меток -->
      <line x1='50' y1='5.000' x2='50.00' y2='10.00'/>
      <line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>
      <line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>
      <line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>
      <line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>
      <line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>
      <line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>
      <line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>
      <line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>
      <line x1='5.000' y1='50.00' x2='10.00' y2='50.00'/>
      <line x1='11.03' y1='27.50' x2='15.36' y2='30.00'/>
      <line x1='27.50' y1='11.03' x2='30.00' y2='15.36'/>
    </g>
    <g id="numbers"> <!-- Числа в основных направлениях -->
      <text x="50" y="18">12</text><text x="85" y="53">3</text>
      <text x="50" y="88">6</text><text x="15" y="53">9</text>
    </g>
    <!-- Нарисовать стрелки, указывающие вверх. Они вращаются сценарием. -->
    <g id="hands" filter="url(#shadow)"> <!-- Добавить тени к стрелкам -->
      <line id="hourhand" x1="50" y1="50" x2="50" y2="24"/>
      <line id="minutehand" x1="50" y1="50" x2="50" y2="20"/>
    </g>
  </svg>
</body>
</html>
```

21.4. Создание графики с помощью элемента <canvas>

Элемент `<canvas>` не имеет собственного визуального представления, но он создает поверхность для рисования внутри документа и предоставляет сценариям на языке JavaScript мощные средства рисования. Элемент `<canvas>` стандартизован спецификацией HTML5, но существует дольше его. Впервые он был реализован компанией Apple в броузере Safari 1.3 и поддерживался броузерами Firefox, начиная с версии 1.5, и Opera, начиная с версии 9. Он также поддерживается всеми версиями Chrome. Элемент `<canvas>` не поддерживался броузером IE до версии IE9, но он с успехом имитировался в IE6, 7 и 8 с помощью свободно распространяемого проекта ExplorerCanvas, домашняя страница которого находится по адресу *http://code.google.com/p/explorercanvas/*.

Существенное отличие между элементом `<canvas>` и технологией SVG заключается в том, что при использовании элемента `<canvas>` изображения формируются вызовами методов, в то время как при использовании формата SVG изображения описываются в виде дерева XML-элементов. Функционально эти два подхода эквивалентны: любой из них может моделироваться с использованием другого. Однако внешне они совершенно отличаются, и каждый из них имеет свои сильные и слабые стороны. Например, из SVG-рисунков легко можно удалять элементы. Чтобы удалить элемент из аналогичного рисунка, созданного в элементе `<canvas>`, обычно требуется полностью ликвидировать рисунок, а затем создать его заново. Поскольку прикладной интерфейс `Canvas` основан на синтаксисе JavaScript, а реализация рисунков с его помощью получается более компактной (чем при использовании формата SVG), я решил описать его в этой книге. Подробные сведения вы найдете в части IV книги в справочных статьях `Canvas`, `CanvasRenderingContext2D` и родственных им.

Трехмерная графика в элементе <canvas>

На момент написания этих строк производители броузеров уже приступили к реализации прикладного интерфейса рисования трехмерной графики в элементе `<canvas>`. Этот прикладной интерфейс называется WebGL и является связующим звеном между JavaScript и стандартным прикладным интерфейсом OpenGL. Чтобы получить объект контекста для рисования трехмерной графики, методу `getContext()` элемента `<canvas>` следует передать строку «webgl». WebGL – это обширный, сложный и низкоуровневый прикладной интерфейс, и он не описывается в этой книге: веб-разработчики, скорее всего, предпочтут использовать вспомогательные библиотеки, основанные на WebGL, чем непосредственно сам прикладной интерфейс WebGL.

Большая часть прикладного интерфейса `Canvas` определена не в элементе `<canvas>`, а в объекте «контекста рисования», получить который можно методом `getContext()` элемента, играющего роль «холста». Вызов метода `getContext()` с аргументом «2d» возвращает объект `CanvasRenderingContext2D`, который можно использовать для рисования двухмерной графики. Важно понимать, что элемент `<canvas>` и объект контекста рисования – это два совершенно разных объекта. Поскольку класс объекта контекста имеет такое длинное имя, я редко буду ссылаться на объект `CanvasRenderingContext2D` по имени, а буду просто называть его «объектом контекста». Аналогично, когда я буду употреблять термин «прикладной интерфейс Canvas», я буду подразумевать «методы объекта `CanvasRenderingContext2D`».

Ниже приводится HTML-страница, которая может служить простым примером использования прикладного интерфейса `Canvas`. Сценарий в ней рисует красный квадрат и голубой круг в элементе `<canvas>`, как показано на рис. 21.2:

```
<body>
Это красный квадрат: <canvas id="square" width=10 height=10></canvas>.
Это голубой круг: <canvas id="circle" width=10 height=10></canvas>.
<script>
var canvas = document.getElementById("square"); // Найти первый элемент canvas
var context = canvas.getContext("2d");          // Получить 2-мерный контекст
context.fillStyle = "#f00";                     // Цвет заливки - красный
context.fillRect(0,0,10,10);                    // Залить квадрат

canvas = document.getElementById("circle");     // Второй элемент canvas
context = canvas.getContext("2d");              // Получить его контекст
context.beginPath();                            // Начать новый "контур"
context.arc(5, 5, 5, 0, 2*Math.PI, true);       // Добавить круг
context.fillStyle = "#00f";                     // Цвет заливки - синий
context.fill();                                 // Залить круг
</script>
</body>
```

Мы видели, что грамматика SVG позволяет описывать сложные фигуры из прямых отрезков и кривых линий, которые могут быть нарисованы или залиты цветом. В прикладном интерфейсе объекта `Canvas` тоже используется понятие контура. Однако контур в данном случае описывается не как строка из символов и чисел, а как последовательность вызовов методов, таких как `beginPath()` и `arc()`, использованных в примере выше. После того как контур будет определен, к нему можно применять различные операции, выполняемые такими методами, как `fill()`. Особенности выполнения операций определяются различными свойствами объекта контекста, такими как `fillStyle`. В следующих подразделах рассказывается:

- Как определять фигуры, как рисовать контуры фигур и как выполнять заливку внутренней области фигур.
- Как устанавливать и читать значение графических атрибутов объекта контекста элемента `<canvas>` и как сохранять и восстанавливать текущие значения этих атрибутов.
- О размерах холста, системе координат по умолчанию элемента `<canvas>` и о том, как выполнять трансформации этой системы координат.
- О различных методах рисования кривых объекта `Canvas`.
- О некоторых специализированных вспомогательных методах рисования прямоугольников.
- Как определять цвета, как работать с прозрачностью и как рисовать градиенты и выполнять заливку шаблонными изображениями.
- Об атрибутах, определяющих толщину линий и внешний вид концов линий и вершин многоугольников.
- Как рисовать текст в элементе `<canvas>`.
- Как ограничивать область холста, чтобы рисование не выполнялось за пределами указанной области.
- Как добавлять тени к фигурам.
- Как рисовать (и при необходимости масштабировать) изображения в элементе `<canvas>` и как извлекать содержимое этого элемента в виде графического изображения.
- Как управлять созданием составных изображений, когда вновь добавляемые (просвечивающие) пикселы объединяются с существующими пикселами.
- Как получать и записывать красную, зеленую, синюю составляющие цвета и степень прозрачности пикселов в элементе `<canvas>`.
- Как определить, возникло ли событие мыши, когда ее указатель находился над нарисованным изображением в элементе `<canvas>`.

В конце этого раздела будет представлен практический пример, в котором элементы `<canvas>` будут использоваться для отображения небольших внутристрочных диаграмм (sparklines).

Во многих примерах работы с элементом `<canvas>`, которые следуют ниже, используется переменная `c`. Эта переменная хранит объект `CanvasRenderingContext2D` элемента `<canvas>`, но инициализация этой переменной в самих примерах обычно не показана. Если у вас появится желание опробовать эти примеры, добавьте разметку HTML, определяющую элемент `<canvas>` с соответствующими атрибутами `width` и `height`, и следующий программный код, инициализирующий переменную `c`:

```
var canvas = document.getElementById("my_canvas_id");
var c = canvas.getContext('2d');
```

Рисунки, которые встретятся вам далее, были созданы сценариями JavaScript, использующими элемент <canvas> – обычно с очень большими размерами, чтобы создать изображения с высоким разрешением, пригодные для печати.

21.4.1. Рисование линий и заливка многоугольников

Чтобы нарисовать прямые линии в элементе <canvas> и залить внутреннюю область замкнутой фигуры, образуемой этими линиями, необходимо сначала определить *контур* (path). Контур – это последовательность из одного или более фрагментов контура. Фрагмент контура – это последовательность из двух или более точек, соединенных прямыми линиями (или, как будет показано ниже, кривыми). Создается новый контур с помощью метода `beginPath()`. А фрагмент контура – с помощью метода `moveTo()`. После установки начальной точки фрагмента контура вызовом метода `moveTo()` можно соединить эту точку с новой точкой прямой линией, вызвав метод `lineTo()`. Следующий фрагмент определяет контур, состоящий из двух прямых линий:

```
c.beginPath();      // Новый контур
c.moveTo(100, 100); // Новый фрагмент контура с начальной точкой (100,100)
c.lineTo(200, 200); // Добавить линию, соединяющую точки (100,100) и (200,200)
c.lineTo(100, 200); // Добавить линию, соединяющую точки (200,200) и (100,200)
```

Фрагмент выше просто определяет контур – он ничего не рисует. Чтобы фактически нарисовать две линии, следует вызвать метод `stroke()`, а чтобы залить область, ограниченную этими линиями, следует вызвать метод `fill()`:

```
c.fill();           // Залить область треугольника
c.stroke();         // Нарисовать две стороны треугольника
```

Фрагмент выше (плюс некоторый дополнительный программный код, устанавливающий толщину линий и цвет заливки) воспроизводит рисунок, изображенный на рис. 21.5.

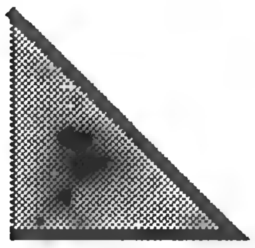


Рис. 21.5. Простой путь, нарисованный и залитый

Обратите внимание, что фрагмент контура, определяемый выше, является «открытым». Он содержит всего две прямые линии, и его конечная точка не совпадает с начальной точкой. То есть он образует незамкнутую область. Метод `fill()` выполняет заливку открытых фрагментов контуров, как если бы конечная и начальная точка фрагмента контура были соединены прямой линией. Именно поэтому пример выше выполняет заливку треугольной области, но рисует только две стороны этого треугольника.

Если бы потребовалось нарисовать все три стороны треугольника выше, можно было бы вызвать метод `closePath()`, чтобы соединить конечную и начальную точки фрагмента контура. (Можно было бы также вызвать метод `lineTo(100,100)`, но в этом случае получились бы три прямые линии с общей начальной и конечной

точками, не образующие в действительности замкнутый фрагмент контура. При рисовании толстыми линиями результат визуально выглядит лучше, если используется метод `closePath()`.)

Следует сделать еще два важных замечания, касающиеся методов `stroke()` и `fill()`. Во-первых, оба метода оперируют всеми элементами в текущем контуре. Допустим, что в примере выше был добавлен еще один фрагмент контура:

```
c.moveTo(300,100); // Новый фрагмент контура с начальной точкой (300,100);
c.lineTo(300,200); // Нарисовать вертикальную линию вниз до точки (300,200);
```

Если затем вызвать метод `stroke()`, получились бы две соединенные вместе стороны треугольника и не связанная с ними вертикальная линия.

Во-вторых, обратите внимание, что методы `stroke()` и `fill()` никогда не изменяют текущий контур: можно вызвать метод `fill()`, и от этого контур никуда не денется, когда вслед за этим будет вызван метод `stroke()`. Когда вы закончили работу с текущим контуром и приступаете к работе с новым контуром, нужно не забыть вызывать метод `beginPath()`. В противном случае вы просто добавите новый фрагмент контура в существующий контур, и старые фрагменты контура будут рисоваться снова и снова.

Пример 21.4 содержит определение функции рисования правильных многоугольников и демонстрирует использование методов `moveTo()`, `lineTo()` и `closePath()` для определения фрагментов контура и методов `fill()` и `stroke()` для рисования контуров. Он воспроизводит рисунок, изображенный на рис. 21.6.

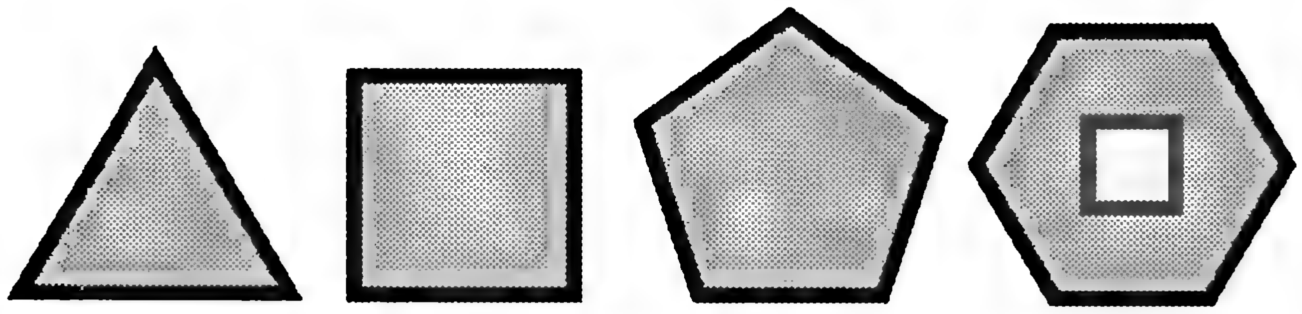


Рис. 21.6. Правильные многоугольники

Пример 21.4. Рисование правильных многоугольников с помощью методов moveTo(), lineTo() и closePath()

```
// Определяет правильный многоугольник с n сторонами, вписанный в окружность с центром
// в точке (x,y) и радиусом r. Вершины многоугольника находятся на окружности,
// на равном удалении друг от друга. Первая вершина помещается в верхнюю точку
// окружности или со смещением на указанный угол angle. Поворот выполняется
// по часовой стрелке, если в последнем аргументе не передать значение true.
function polygon(c,n,x,y,r,angle,counterclockwise) {
    angle = angle || 0;
    counterclockwise = counterclockwise || false;
    c.moveTo(x + r*Math.sin(angle),  // Новый фрагмент контура
             y - r*Math.cos(angle)); // Исп. тригонометрию для выч. координат
    var delta = 2*Math.PI/n;         // Угловое расстояние между вершинами
    for(var i = 1; i < n; i++) {     // Для каждой из оставшихся вершин
        angle += counterclockwise?-delta:delta; // Скорректировать угол
        c.lineTo(x + r*Math.sin(angle),         // Линия к след. вершине
                 y - r*Math.cos(angle));
    }
    c.closePath(); // Соединить первую вершину с последней
}

// Создать новый контур и добавить фрагменты контура, соответствующие многоугольникам
c.beginPath();
polygon(c, 3, 50, 70, 50);                    // Треугольник
polygon(c, 4, 150, 60, 50, Math.PI/4);        // Квадрат
polygon(c, 5, 255, 55, 50);                   // Пятиугольник
polygon(c, 6, 365, 53, 50, Math.PI/6);        // Шестиугольник
polygon(c, 4, 365, 53, 20, Math.PI/4, true); // Квадрат в шестиугольнике

// Установить некоторые свойства, определяющие внешний вид рисунка
c.fillStyle = "#ccc";   // Светло-серый фон внутренних областей
c.strokeStyle = "#008"; // темно-синие контуры
c.lineWidth = 5;        // толщиной пять пикселов.

// Нарисовать все многоугольники (каждый создается в виде отдельного фрагмента контура)
// следующими вызовами
c.fill();   // Залить фигуры
c.stroke(); // И нарисовать контур
```

Обратите внимание, что в этом примере внутри шестиугольника рисуется квадрат. Квадрат и шестиугольник являются отдельными фрагментами контура, но они перекрываются. Когда это происходит (или когда один фрагмент контура пересекается с самим собой), элементу `<canvas>` приходится выяснять, какая область является внутренней для фрагмента контура, а какая – внешней. Для этого элемент `<canvas>` использует алгоритм, известный как «правило ненулевого числа оборотов» («nonzero winding rule»). В данном случае внутренняя область квадрата не заливается светло-серым цветом, потому что квадрат и шестиугольник рисовались в противоположных направлениях: вершины шестиугольника соединялись линиями в направлении по часовой стрелке, а вершины квадрата – против часовой стрелки. Если бы рисование квадрата также выполнялось в направлении по часовой стрелке, метод `fill()` залил бы внутреннюю область квадрата.

Правило ненулевого числа оборотов

Чтобы определить, находится ли точка P внутри контура, используя правило правило ненулевого числа оборотов, представьте луч, исходящий из точки P в любом направлении и уходящий в бесконечность (или, если говорить практическим языком, до некоторой точки за пределами фигуры, образуемой рассматриваемым контуром). Теперь инициализируйте счетчик нулевым значением и проследите за всеми пересечениями контура с лучом. Каждый раз, когда луч пересекает контур, рисуемый в направлении по часовой стрелке, увеличивайте счетчик на единицу. Каждый раз, когда луч пересекает контур, рисуемый в направлении против часовой стрелки, уменьшайте счетчик на единицу. Если после учета всех пересечений в счетчике получается значение, не равное нулю, следовательно, точка P находится внутри контура. Если счетчик оказывается равным нулю, следовательно, точка P находится снаружи.

21.4.2. Графические атрибуты

В примере 21.4 устанавливаются свойства `fillStyle`, `strokeStyle` и `lineWidth` объекта контекста элемента `<canvas>`. Эти свойства являются графическими атрибутами, определяющими цвет, используемый методом `fill()`; цвет, используемый методом `stroke()`; и толщину линий, рисуемых методом `stroke()`. Обратите внимание, что эти параметры не передаются методам `fill()` и `stroke()`, а являются общими *графическими свойствами* элемента `<canvas>`. Если определяется метод, рисующий некоторую фигуру, который не устанавливает эти свойства, программа, использующая его, сможет сама определять цвет фигуры, устанавливая свойства `strokeStyle` и `fillStyle` перед вызовом этого метода. Такое отделение графических свойств от команд рисования является фундаментальной особенностью прикладного интерфейса объекта `Canvas` и сродни отделению представления от содержимого, достигаемого за счет применения таблиц стилей CSS к HTML-документам.

Прикладной интерфейс объекта `Canvas` содержит 15 свойств графических атрибутов в объекте `CanvasRenderingContext2D`. Эти свойства перечислены в табл. 21.1 и подробно описываются в соответствующих разделах ниже.

Таблица 21.1. Графические атрибуты прикладного интерфейса объекта Canvas

Свойство	Описание
`fillStyle`	цвет, градиент или шаблон, используемый для заливки
`font`	определение шрифта в формате CSS для команд рисования текста
`globalAlpha`	уровень прозрачности, назначаемый для всех пикселов при рисовании
`globalCompositeOperation`	способ объединения новых пикселов с существующими
`lineCap`	форма концов линий
`lineJoin`	форма вершин
`lineWidth`	толщина рисуемых линий
`miterLimit`	максимальная длина острых вершин
`textAlign`	выравнивание текста по горизонтали
`textBaseline`	выравнивание текста по вертикали
`shadowBlur`	четкость теней
`shadowColor`	цвет теней
`shadowOffsetX`	горизонтальное смещение теней
`shadowOffsetY`	вертикальное смещение теней
`strokeStyle`	цвет, градиент или шаблон, используемый для рисования линий

Так как прикладной интерфейс объекта `Canvas` определяет графические атрибуты в объекте контекста, может появиться идея вызвать метод `getContext()` несколько раз, чтобы получить несколько объектов контекста. Если бы это удалось, можно было бы определить для каждого из них различные атрибуты и использовать их как различные кисти разного цвета и разной толщины. К сожалению, элемент `<canvas>` нельзя использовать таким способом. Каждый элемент `<canvas>` имеет только один объект контекста, и каждый вызов метода `getContext()` возвращает один и тот же объект `CanvasRenderingContext2D`.

Тем не менее, несмотря на то что прикладной интерфейс объекта Canvas позволяет определить только один набор графических атрибутов, он предусматривает возможность сохранять текущие графические свойства, чтобы их можно было изменить и позднее легко восстановить прежние значения. Метод save() помещает текущие значения графических свойств в стек. Метод restore() выталкивает их со стека и восстанавливает самые последние сохраненные значения. В множество сохраняемых свойств входят все свойства, перечисленные в табл. 21.1, а также текущее преобразование системы координат и область отсечения (обе особенности рассматриваются ниже). Важно отметить, что текущее определение контура и координаты текущей точки не входят в множество сохраняемых графических свойств и не могут сохраняться и восстанавливаться.

Если вам потребуется больше гибкости, чем может обеспечит простой стек графических свойств, вы можете определить вспомогательные методы, такие как в примере 21.5.

Пример 21.5. Утилиты управления графическими свойствами

```
// Восстанавливает последние сохраненные значения графических свойств,
// но не выталкивает их со стека.
CanvasRenderingContext2D.prototype.revert = function() {
    this.restore(); // Восстановить прежние значения графических свойств.
    this.save();    // Сохранить их снова, чтобы можно было вернуться к ним.
    return this;    // Позволить составление цепочек вызовов методов.
};

// Устанавливает графические атрибуты в соответствии со значениями свойств объекта o.
// Или, при вызове без аргументов, возвращает текущие значения атрибутов в виде объекта.
// Обратите внимание, что этот метод не обслуживает преобразование или область отсечения.
CanvasRenderingContext2D.prototype.attrs = function(o) {
    if (o) {
        for(var a in o)       // Для каждого свойства объекта o
            this[a] = o[a];   // Установить его как графический атрибут
        return this;          // Позволить составление цепочек вызовов методов
    }
    else return {
        fillStyle: this.fillStyle, font: this.font,
        globalAlpha: this.globalAlpha,
        globalCompositeOperation: this.globalCompositeOperation,
        lineCap: this.lineCap, lineJoin: this.lineJoin,
        lineWidth: this.lineWidth, miterLimit: this.miterLimit,
        textAlign: this.textAlign, textBaseline: this.textBaseline,
        shadowBlur: this.shadowBlur, shadowColor: this.shadowColor,
        shadowOffsetX: this.shadowOffsetX, shadowOffsetY: this.shadowOffsetY,
        strokeStyle: this.strokeStyle
    };
};
```

21.4.3. Размеры и система координат холста

Атрибуты width и height элемента <canvas> и соответствующие им свойства width и height объекта Canvas определяют размеры холста. По умолчанию начало системы координат холста (0,0) находится в его левом верхнем углу. Координата X увеличивается в направлении слева направо, а координата Y – сверху вниз. Координаты точек на холсте могут определяться вещественными значениями, и они не будут автоматически округляться до целых – для имитации частично заполненных пикселов объект Canvas использует приемы сглаживания.

Размеры холста являются настолько фундаментальными характеристиками, что они не могут изменяться без полного сброса холста в исходное состояние. Изменение значения свойства width или height объекта Canvas (даже присваивание им текущих значений) вызывает очистку холста, стирание текущего контура и переустановку всех графических атрибутов (включая текущее преобразование и область отсечения) в исходное состояние.

Несмотря на фундаментальность размеров холста, они необязательно совпадают с размерами холста на экране или с количеством пикселов, образующих поверхность для рисования. Размеры холста (а также система координат по умолчанию) измеряются в CSS-пикселах. CSS-пикселы обычно совпадают с обычными пикселами. Однако на устройствах высокого разрешения реализациям разрешено отображать несколько аппаратных пикселов в один CSS-пиксел. Это означает, что размеры прямоугольной области в пикселах, отведенной для холста, могут оказаться больше номинальных размеров холста. Об этом следует помнить при работе с механизмами холста, манипулирующими пикселами (раздел 21.4.14), но в других случаях различия между виртуальными CSS-пикселами и фактическим аппаратными пикселами не оказывают влияния на программный код, выполняющий операции с холстом.

По умолчанию элемент <canvas> отображается на экране с размерами (в CSS-пикселах), указанными в его HTML-атрибутах width и height. Однако, подобно любым другим HTML-элементам, элемент <canvas> может иметь экранные размеры, определяемые CSS-атрибутами стиля width и height. Если экранные размеры холста отличаются от его фактических размеров, пикселы холста автоматически будут масштабироваться в соответствии с экранными размерами, указанными в CSS-атрибутах. Экранные размеры холста никак не влияют на количество CSS- или аппаратных пикселов, зарезервированных в растровом отображении холста, а масштабирование выполняется как обычная операция масштабирования изображений. Если экранные размеры оказываются существенно больше фактических размеров холста, это приводит к появлению мозаичного эффекта. Однако это проблема скорее для художников и никак не влияет на программирование холста.

21.4.4. Преобразование системы координат

Как отмечалось выше, начало системы координат холста по умолчанию находится в левом верхнем углу, значение координаты X увеличивается в направлении слева направо, а координаты Y – сверху вниз. В этой системе координат по умолчанию координаты точки отображаются непосредственно в CSS-пикселы (которые затем отображаются в один или более аппаратных пикселов). Некоторые операции с холстом и атрибутами (такие как извлечение параметров пикселов и установка смещения теней) всегда используют систему координат по умолчанию. Однако помимо системы координат по умолчанию каждый холст имеет в составе графических свойств «текущую матрицу преобразований». Эта матрица определяет текущую систему координат холста. В большинстве операций с холстом, где указываются координаты точки, используется текущая система координат, а не система координат по умолчанию. Текущая матрица преобразований используется для преобразования указанных вами координат в эквивалентные им координаты в системе координат по умолчанию.

Метод `setTransform()` позволяет напрямую определять матрицу преобразований холста, но обычно преобразования системы координат проще определять как последовательность операций смещения, вращения и масштабирования. Влияние этих операций на систему координат холста иллюстрируются на рис. 21.7. Программа, с помощью которой был получен этот рисунок, семь раз подряд использовала одно и то же изображение осей координат. Единственное, что изменялось каждый раз, – это текущее преобразование. Обратите внимание, что преобразования оказывают влияние не только на рисование линий, но и на вывод текста.

Метод `translate()` просто смещает начало системы координат влево, вправо, вверх или вниз. Метод `rotate()` выполняет вращение осей координат по часовой стрелке на указанный угол. (Прикладной интерфейс объекта `Canvas` всегда измеряет углы в радианах. Чтобы преобразовать градусы в радианы, необходимо разделить значение в градусах на 180 и умножить на `Math.PI`.) Метод `scale()` растягивает или сжимает расстояния по оси X или Y.

Передача отрицательного коэффициента масштабирования методу `scale()` переворачивает соответствующую ему ось относительно начала системы координат, создавая зеркальное ее отражение. Именно это преобразование можно наблюдать внизу слева на рис. 21.7: вызов метода `translate()` сместил начало координат в левый нижний угол холста, а вызов метода `scale()` перевернул ось Y так, что значения координаты Y стали увеличиваться в направлении снизу вверх. Такая перевернутая система координат должна быть знакома вам по школьному курсу алгебры, и она может пригодиться для рисования графиков и диаграмм. Отметьте, однако, что текст после такого преобразования очень трудно читать!

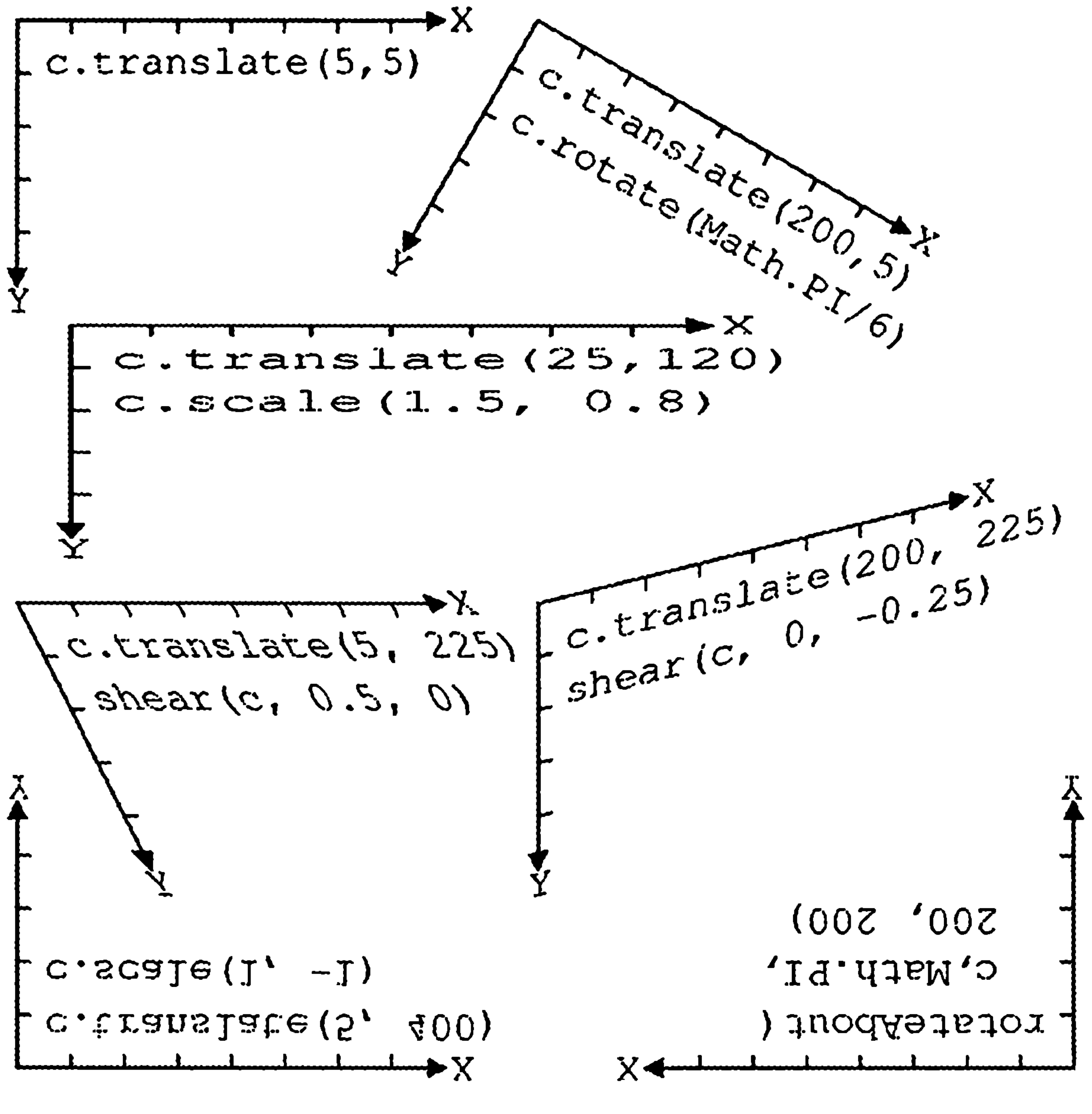


Рис. 21.7. Преобразования системы координат

21.4.4.1. Математический смысл преобразований

На мой взгляд, проще всего разбираться с преобразованиями, имея их геометрическое представление, когда действие методов `translate()`, `rotate()` и `scale()` можно представить в виде преобразований координатных осей, как показано на рис. 21.7. Преобразования можно также представить алгебраически, в виде системы уравнений, отображающих координаты точки (x,y) в преобразованной системе координат в координаты той же точки (x',y') в исходной системе координат.

Вызов метода `c.translate(dx,dy)` можно описать следующими уравнениями:

```
x' = x + dx; // Значение 0 координаты X в новой системе координат
             // соответствует значению dx в старой системе координат
y' = y + dy;
```

Операцию масштабирования также можно представить в виде простых уравнений. Вызов метода `c.scale(sx,sy)` можно описать следующим образом:

```
x' = sx * x;
y' = sy * y;
```

Операция вращения выглядит несколько сложнее. Вызов `c.rotate(a)` описывается следующими тригонометрическими уравнениями:

```
x' = x * cos(a) - y * sin(a);
y' = y * cos(a) + x * sin(a);
```

Обратите внимание, что порядок выполнения преобразований имеет большое значение. Допустим, что изначально используется система координат холста по умолчанию, после чего выполняется смещение и затем масштабирование. Чтобы отобразить координаты точки (x,y) в текущей системе координат обратно в координаты (x'',y'') в системе координат по умолчанию, необходимо сначала применить уравнения масштабирования, чтобы отобразить координаты точки в промежуточные координаты (x', y') точки в смещенной, но не масштабированной системе координат, а затем применить уравнения смещения, чтобы отобразить эти промежуточные координаты точки в координаты (x'',y''). В результате получим следующую систему уравнений:

```
x'' = sx*x + dx;
y'' = sy*y + dy;
```

Если же к исходной системе координат сначала применялся метод `scale()`, а затем `translate()`, мы придем к другой системе уравнений:

```
x'' = sx*(x + dx);
y'' = sy*(y + dy);
```

При использовании алгебраических представлений последовательностей преобразований важно помнить, что в уравнениях они должны следовать в обратном порядке – от последнего преобразования к первому. Однако при использовании геометрических представлений вы работаете с последовательностями преобразований в прямом порядке, от первого к последнему.

Преобразования, поддерживаемые холстом, известны как *аффинные преобразования*. Аффинные преобразования могут изменять расстояния между точками и углы между линиями, но параллельные линии всегда остаются параллельными после любых аффинных преобразований – с помощью аффинных преобразований нельзя, например, создать эффект искажения типа «рыбий глаз». Любое аффинное преобразование можно описать с помощью шести параметров от a до f, как показано в следующих уравнениях:

```
x' = ax + cy + e
y' = bx + dy + f
```

К текущей системе координат можно применять любые преобразования, передавая эти шесть параметров методу `transform()`. На рис. 21.7 показаны два типа преобразований – сдвиг и вращение вокруг указанной точки – которые можно реализовать с помощью метода `transform()`, как показано ниже:

```
// Сдвиг:
// x' = x + kx*y;
// y' = y + ky*x;
function shear(c,kx,ky) { c.transform(1, ky, kx, 1, 0, 0); }

// Вращение на theta радиан по часовой стрелке вокруг точки (x,y). Это преобразование
// можно выполнить с помощью последовательности вызовов методов translate,rotate,translate
function rotateAbout(c,theta,x,y) {
    var ct = Math.cos(theta), st = Math.sin(theta);
    c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);
}
```

Метод `setTransform()` принимает те же аргументы, что и метод `transform()`, но вместо преобразования текущей системы координат он выполняет преобразование системы координат по умолчанию и делает результат новой текущей системой координат. Метод `setTransform()` удобно использовать для временного возврата к системе координат по умолчанию:

```
c.save();                      // Сохранить текущую систему координат
c.setTransform(1,0,0,1,0,0); // Вернуться к системе координат по умолчанию
// Выполнить операции с использованием координат по умолчанию CSS-пикселов
c.restore();                   // Восстановить сохраненную систему координат
```

21.4.4.2. Примеры преобразований

Пример 21.6 демонстрирует мощь, которую дает возможность преобразования системы координат, где за счет рекурсивного применения методов `translate()`, `rotate()` и `scale()` реализовано рисование фракталов – снежинок Коха. Результат работы этого примера представлен на рис. 21.8, где показаны снежинки Коха с количеством уровней рекурсии 0, 1, 2, 3 и 4.

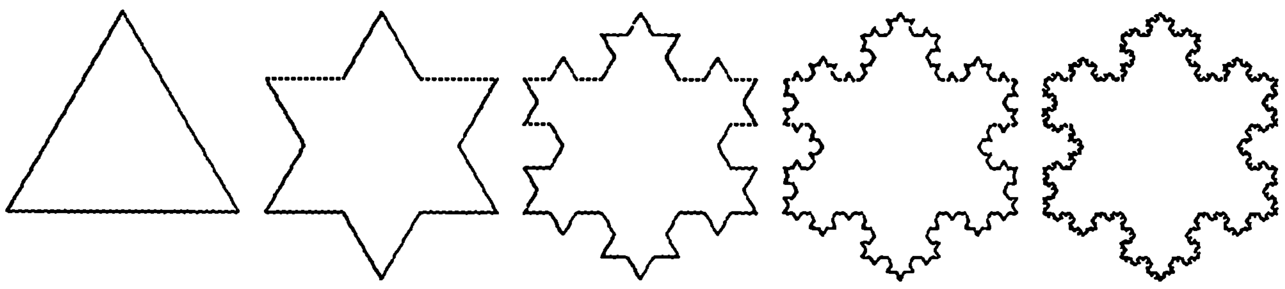


Рис. 21.8. Снежинки Коха

Эти фигуры воспроизводятся весьма изящным программным кодом, но в нем используются рекурсивные преобразования системы координат, что делает его сложным для понимания. Даже если вы не собираетесь углубляться в изучение всех тонкостей примера, обратите все же внимание, что в нем имеется всего один вызов метода `lineTo()`. Каждый отдельный сегмент на рис. 21.8 рисуется следующим образом:

```
c.lineTo(len, 0);
```

Значение переменной `len` не изменяется в ходе выполнения программы, поэтому позиция, ориентация и длина каждой линии определяется операциями смещения, вращения и масштабирования.

Пример 21.6. Рисование снежинок Коха посредством преобразований системы координат

```
var deg = Math.PI/180;  // Для преобразования градусов в радианы

// Рисует n-уровневый фрактал снежинки Коха в контексте холста c, левый нижний угол
// которого имеет координаты (x,y), а длина стороны равна len.
function snowflake(c, n, x, y, len) {
    c.save();           // Сохранить текущее преобразование
    c.translate(x,y);   // Сместить начало координат в начальную точку
    c.moveTo(0,0);      // Новый фрагмент контура в новом начале координат
    leg(n);             // Нарисовать первую ветвь снежинки
    c.rotate(-120*deg); // Поворот на 120 градусов против часовой стрелки
    leg(n);             // Нарисовать вторую ветвь
    c.rotate(-120*deg); // Поворот
    leg(n);             // Нарисовать последнюю ветвь
    c.closePath();      // Замкнуть фрагмент контура
    c.restore();        // Восстановить прежнее преобразование

    // Рисует одну ветвь n-уровневой снежинки Коха. Эта функция оставляет
    // текущую позицию в конце нарисованной ветви и смещает начало координат так,
    // что текущая точка оказывается в позиции (0,0).
    // Это означает, что после рисования ветви можно вызвать rotate().
    function leg(n) {
        c.save();               // Сохранить текущее преобразование
        if (n == 0) {           // Нерекурсивный случай:
            c.lineTo(len, 0);   // Просто нарисовать горизонтальную линию
        }                       //
        else {                  // Рекурсивный случай: 4 подветви вида: ‾\/‾
            c.scale(1/3,1/3);   // Подветви в 3 раза меньше этой ветви
            leg(n-1);           // Рекурсия для первой подветви
            c.rotate(60*deg);   // Поворот на 60 градусов по часовой стрелке
            leg(n-1);           // Вторая подветвь
            c.rotate(-120*deg); // Поворот на 120 градусов назад
            leg(n-1);           // Третья подветвь
            c.rotate(60*deg);   // Поворот обратно к началу
            leg(n-1);           // Последняя подветвь
        }
        c.restore();            // Восстановить преобразование
        c.translate(len, 0);    // Но сместить в конец ветви (0,0)
    }
}

snowflake(c,0,5,115,125);    // Снежинка нулевого уровня является
                             // равносторонним треугольником
snowflake(c,1,145,115,125);  // Снежинка первого уровня - шестиконечная звезда
snowflake(c,2,285,115,125); // и так далее.
snowflake(c,3,425,115,125);
snowflake(c,4,565,115,125); // Снежинка четвертого уровня выглядит как снежинка!
c.stroke();                 // Нарисовать этот очень сложный контур
```

21.4.5. Рисование и заливка кривых

Контур – это последовательность фрагментов контура, а фрагмент контура – это последовательность точек, соединенных линиями. В контурах, которые определялись в разделе 21.4.1, эти точки соединялись прямыми линиями, но это не обязательно должно быть так. Объект `CanvasRenderingContext2D` определяет несколько методов, которые добавляют во фрагмент контура новую точку и соединяют ее кривой с текущей точкой:

`arc()`

Этот метод добавляет во фрагмент контура дугу. Он соединяет текущую точку с началом дуги прямой линией и затем соединяет начало дуги с концом дуги сегментом окружности, при этом конечная точка дуги становится новой текущей точкой. Дуга определяется шестью параметрами: координаты X и Y центра окружности, радиус окружности, угол начала и конца дуги и направление (по часовой стрелке или против часовой стрелки) рисования дуги между этими двумя углами.

`arcTo()`

Этот метод так же рисует прямую линию и круговую дугу, как и метод `arc()`, но при его использовании дуга определяется другим набором параметров. Аргументы метода `arcTo()` определяют точки P1 и P2 и радиус. Дуга, добавляемая в контур, имеет указанный радиус и строится так, что линии, соединяющие текущую точку с точкой P1 и точку P1 с точкой P2, являются касательными к ней. Такой необычный, на первый взгляд, способ определения дуг в действительности является весьма удобным при рисовании закругленных углов. Если указать радиус равный 0, этот метод просто нарисует прямую линию, соединяющую текущую точку и точку P1. Однако если указан ненулевой радиус, он нарисует прямую линию от текущей точки в направлении точки P1 до начала дуги, затем начнет рисовать круговую дугу, пока направление рисования не совпадет с направлением на точку P2.

`bezierCurveTo()`

Этот метод добавит во фрагмент контура новую точку P и соединит ее с текущей точкой кубической кривой Безье. Форма кривой определяется двумя «контрольными точками» C1 и C2. В начале кривой (в текущей точке) рисование начинается в направлении точки C1. В свой конец (в точке P) кривая приходит в направлении из точки C2. Между этими точками кривая плавно изгибается. Точка P становится новой текущей точкой для фрагмента контура.

`quadraticCurveTo()`

Этот метод похож на метод `bezierCurveTo()`, но рисует квадратичные кривые Безье и имеет всего одну контрольную точку.

С помощью этих методов можно рисовать контуры, подобные тем, что изображены на рис. 21.9.

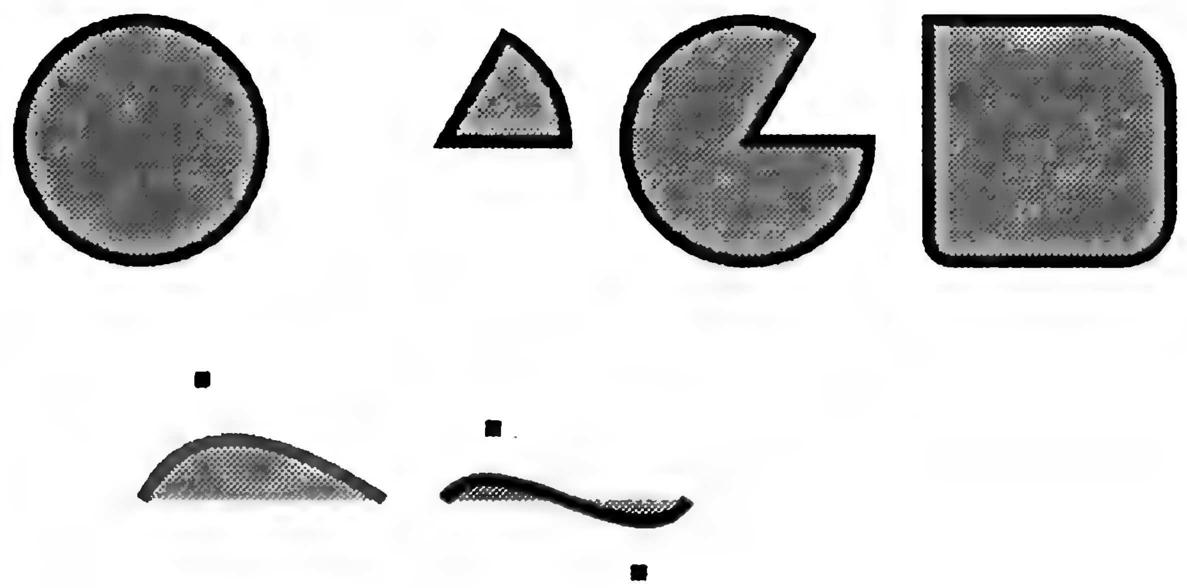


Рис. 21.9. Контуры, состоящие из кривых

В примере 21.7 представлен программный код, с помощью которого было создано изображение на рис. 21.9. Методы, используемые в этом примере, являются одними из самых сложных в прикладном интерфейсе объекта Canvas. **Полное описание методов и их аргументов приводится в справочном разделе книги.**

Пример 21.7. Добавление кривых в контур

```
// Вспомогательная функция для преобразования градусов в радианы
function rads(x) { return Math.PI*x/180; }

// Нарисовать окружность. Используйте масштабирование и вращение, если требуется
// получить эллипс. Здесь не используется текущая точка, поэтому окружность рисуется
// без прямой линии, соединяющей текущую точку с началом окружности.
c.beginPath();
c.arc(75,100,50,             // Центр в точке (75,100), радиус 50
      0,rads(360),false);    // По часовой стрелке от 0 до 360 градусов

// Нарисовать сектор. Углы откладываются по часовой стрелке от положительной оси x.
// Обратите внимание, что метод arc() добавляет линию от текущей точки к началу дуги.
c.moveTo(200, 100);          // Перейти в центр окружности
c.arc(200, 100, 50,          // Центр окружности и радиус
      rads(-60), rads(0),    // Начальный угол -60 градусов, конечный 0 градусов
      false);                // false означает по часовой стрелке
c.closePath();               // Добавить прямую линю к центру окружности

// Тот же сектор, в противоположном направлении
c.moveTo(325, 100);
c.arc(325, 100, 50, rads(-60), rads(0), true); // Против часовой стрелки
c.closePath();

// Использовать arcTo() для закругления углов. Здесь рисуется квадрат с верхним левым
// углом в точке (400,50), с закруглениями углов дугами с разными радиусами.
c.moveTo(450, 50);           // Середина верхней стороны.
c.arcTo(500,50,500,150,30);  // Часть верхней стороны и правый верхний угол.
c.arcTo(500,150,400,150,20); // Добавить правую сторону и правый нижний угол.
c.arcTo(400,150,400,50,10);  // Добавить нижнюю сторону и левый нижний угол.
c.arcTo(400,50,500,50,0);    // Добавить левую сторону и левый верхний угол.
c.closePath(); // Замкнуть контур, чтобы добавить остаток верхней стороны.

// Квадратичная кривая Безье: одна контрольная точка
c.moveTo(75, 250);                      // Начало в точке (75,250)
c.quadraticCurveTo(100,200, 175, 250); // Соединить с точкой (175,250)
c.fillRect(100-3,200-3,6,6);            // Метка контрольной точки (100,200)
```

```
// Кубическая кривая Безье
c.moveTo(200, 250);                        // Начало в точке (200,250)
c.bezierCurveTo(220,220,280,280,300,250); // Соединить с точкой (300,250)
c.fillRect(220-3,220-3,6,6);               // Метки контрольных точек
c.fillRect(280-3,280-3,6,6);

// Определить некоторые графические атрибуты и нарисовать кривые
c.fillStyle = "#aaa"; // Серый цвет заливки
c.lineWidth = 5;      // Черные (по умолчанию) линии толщиной 5 пикселов
c.fill();             // Залить фигуры
c.stroke();           // Нарисовать контуры
```

21.4.6. Прямоугольники

Объект `CanvasRenderingContext2D` определяет четыре метода рисования прямоугольников. Один из них, `fillRect()`, использовался в примере 21.7 для создания меток контрольных точек кривых Безье. Все четыре метода рисования прямоугольников принимают два аргумента, определяющих координаты одного угла прямоугольника, и два аргумента, определяющих ширину и высоту. Обычно указывается верхний левый угол и положительные значения ширины и высоты, но можно также указать другие углы и передать отрицательные размеры.

Метод `fillRect()` выполняет заливку внутренней области прямоугольника в соответствии со значением атрибута `fillStyle`. Метод `strokeRect()` рисует контур прямоугольника, используя текущее значение атрибута `strokeStyle` и других атрибутов линий. Метод `clearRect()` подобен методу `fillRect()`, но он игнорирует текущее значение стиля заливки и заполняет прямоугольник прозрачными черными пикселами (цвет по умолчанию всех пустых холстов). Важно отметить, что все эти три метода не оказывают влияния ни на текущий контур, ни на текущую точку внутри этого контура.

Последний метод рисования прямоугольников называется `rect()`, и он изменяет текущий контур: он добавляет указанный прямоугольник в виде отдельного фрагмента контура. Подобно другим методам определения контуров, сам по себе он не производит ни заливку, ни рисование контура.

21.4.7. Цвет, прозрачность, градиенты и шаблоны

Атрибуты `strokeStyle` и `fillStyle` определяют параметры рисования линий и заливки областей. Чаще всего эти атрибуты используются, чтобы определить непрозрачный или полупрозрачный цвет, но им также можно присваивать объекты `CanvasPattern` и `CanvasGradient`, чтобы рисование или заливка выполнялась с использованием повторяющегося изображения или линейного или радиального градиента. Кроме того, можно воспользоваться свойством `globalAlpha`, чтобы сделать полупрозрачным все, что будет рисоваться.

Чтобы определить сплошной цвет, можно использовать имена цветов, определяемые стандартом HTML4[1], или использовать строки в формате CSS:

```
context.strokeStyle = "blue";   // Рисовать синие линии
context.fillStyle = "#aaa";     // Заливку выполнять серым цветом
```

---

[1] Aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white и yellow.

По умолчанию свойства `strokeStyle` и `fillStyle` имеют значение «#000000», соответствующее непрозрачному черному цвету. Текущие броузеры поддерживают цвета CSS3 и позволяют использовать форматы RGB, RGBA, HSL и HSLA определения цветов вдобавок к базовому формату RGB. Например:

```
var colors = [
    "#f44",                    // Шестнадцатеричное значение RGB: красный
    "#44ff44",                 // Шестнадцатеричное значение RRGGBB: зеленый
    "rgb(60, 60, 255)",        // RGB в виде целых 0-255: синий
    "rgb(100%, 25%, 100%)",    // RGB в виде процентов: пурпурный
    "rgba(100%, 25%, 100%, 0.5)",// RGB плюс альфа 0-1: полупрозрачный пурпурный
    "rgba(0,0,0,0)",           // Совершенно прозрачный черный
    "transparent",             // Синоним предыдущего цвета
    "hsl(60, 100%, 50%)",      // Насыщенный желтый
    "hsl(60, 75%, 50%)",       // Менее насыщенный желтый
    "hsl(60, 100%, 75%)",      // Насыщенный желтый, немного светлее
    "hsl(60, 100%, 25%)",      // Насыщенный желтый, немного темнее
    "hsla(60, 100%, 50%, 0.5)", // Насыщенный желтый, прозрачный на 50%
];
```

При использовании формата HSL цвет описывается тремя числами, определяющими тон (hue), насыщенность (saturation) и яркость (lightness). Тон (hue) – это величина угла в цветовом круге. Значение 0 соответствует красному цвету, 60 – желтому, 120 – зеленому, 180 – бирюзовому, 240 – синему, 300 – сиреневому и 360 – опять красному. Насыщенность описывает интенсивность цвета и определяется в процентах. Цвета с насыщенностью 0% являются оттенками серого. Яркость описывает степень яркости цвета и также определяется в процентах. Любой цвет в формате HSL со 100-процентной яркостью является белым цветом, а любой цвет с яркостью 0% – черным. В формате HSLA цвет описывается так же, как в формате HSL, но с дополнительным значением альфа-канала, которое изменяется в диапазоне от 0.0 (прозрачный) до 1.0 (непрозрачный).

Если необходимо использовать полупрозрачные цвета, но нежелательно явно указывать значение альфа-канала для каждого цвета, или если необходимо добавить полупрозрачность к непрозрачному изображению или шаблону (например), требуемое значение непрозрачности можно присвоить свойству `globalAlpha`. Значение альфа-канала каждого пиксела, рисуемого вами, будет умножаться на значение свойства `globalAlpha`. По умолчанию это свойство имеет значение 1 и не добавляет прозрачности. Если свойству `globalAlpha` присвоить значение 0, все нарисованное вами станет полностью прозрачным и на холсте ничего не будет видно. Если присвоить этому свойству значение 0.5, непрозрачные пикселы станут наполовину прозрачными. А пикселы, степень непрозрачности которых была равна 50%, станут непрозрачными на 25%. Изменение значения свойства `globalAlpha` оказывает влияние на степень непрозрачности всех пикселов, поэтому вам, вероятно, потребуется учесть, как эти пикселы объединяются (или «составляются») с пикселами, поверх которых они нарисованы – режимы объединения, поддерживаемые объектом `Canvas`, описываются в разделе 21.4.13.

Вместо сплошного цвета (пусть и полупрозрачного), заливку и рисование контуров можно также выполнять с использованием градиентов и повторяющихся изображений. На рис. 21.10 изображен прямоугольник, контур которого нарисован толстыми линиями с использованием шаблонного изображения поверх заливки линейным градиентом и под заливкой радиальным градиентом. Ниже описывается, как было реализовано рисование шаблонным изображением и заливка градиентами.

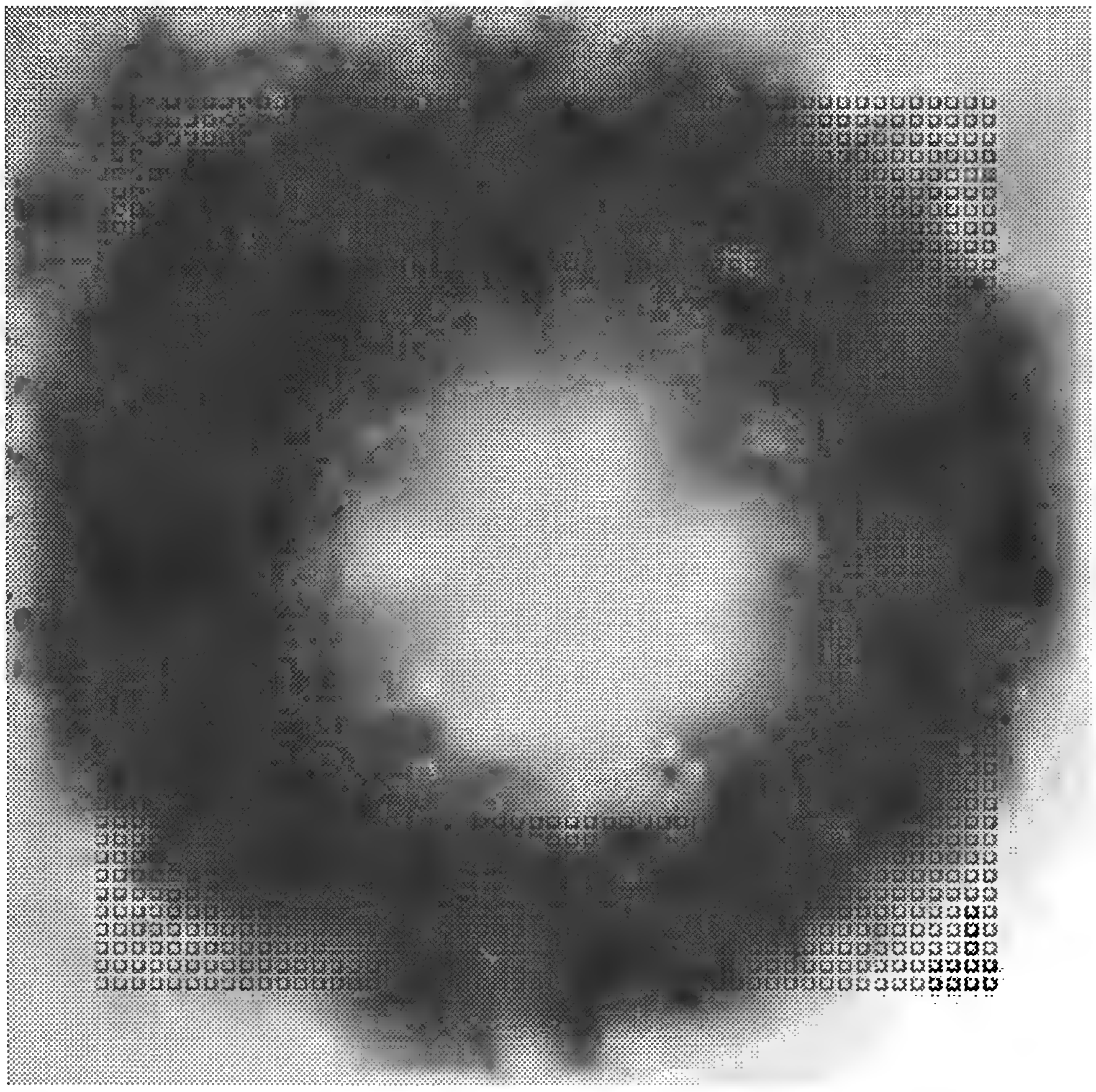


Рис. 21.10. Рисование шаблоном и заливка градиентом

Чтобы выполнить заливку или рисование с применением шаблонного изображения вместо цвета, следует присвоить свойству `fillStyle` или `strokeStyle` объект `CanvasPattern`, возвращаемый методом `createPattern()` объекта контекста:

```
var image = document.getElementById("myimage");
c.fillStyle = c.createPattern(image, "repeat");
```

Первый аргумент метода `createPattern()` определяет изображение, которое будет использовано как шаблон. Это должен быть элемент документа `<img>`, `<canvas>` или `<video>` (или объект `Image`, созданный конструктором `Image()`). Во втором аргументе обычно передается строка «repeat», если требуется повторять изображение при заполнении, независимо от его размера, но можно также использовать значения «repeat-x», «repeat-y» или «no-repeat».

Обратите внимание, что в качестве шаблонного изображения для одного элемента `<canvas>` можно использовать другой элемент `<canvas>` (даже если он не включен в состав документа и невидим на экране):

```
var offscreen = document.createElement("canvas"); // Невидимый холст
offscreen.width = offscreen.height = 10;          // Установить его размеры
offscreen.getContext("2d").strokeRect(0,0,6,6);   // Получить контекст
                                                  // и нарисовать прямоугольник
var pattern = c.createPattern(offscreen,"repeat");// И использовать как шаблон
```

Чтобы выполнить заливку (или нарисовать контур) градиентом, следует присвоить свойству `fillStyle` (или `strokeStyle`) объект `CanvasGradient`, возвращаемый методом `createLinearGradient()` или `createRadialGradient()` объекта контекста. Создание градиентов выполняется в несколько этапов, и в использовании они несколько сложнее, чем шаблонные изображения.

Первый этап – создание объекта `CanvasGradient`. В качестве аргументов методу `createLinearGradient()` передаются координаты двух точек, определяющих линию (она необязательно должна быть горизонтальной или вертикальной), вдоль которой будет изменяться цвет. Аргументы метода `createRadialGradient()` определяют центры и радиусы двух окружностей. (Они необязательно должны быть концентрическими, но первая окружность обычно полностью располагается внутри второй.) Области внутри малой окружности и за пределами большой окружности будут заполняться сплошным цветом, а область между окружностями – градиентом.

После того как объект `CanvasGradient` создан и определены области холста для заливки, необходимо определить цвета градиента вызовом метода `addColorStop()` объекта `CanvasGradient`. В первом аргументе этому методу передается число в диапазоне от 0.0 до 1.0. Во втором – цвет в формате, поддерживаемом CSS. Этот метод должен вызываться как минимум два раза, чтобы определить простой градиент, но его можно вызвать большее число раз. Цвет, соответствующий значению 0.0, будет использоваться в начале градиента, а цвет, соответствующий значению 1.0, – в конце. Если вы решите указать дополнительные цвета, они будут использоваться в промежуточных позициях градиента. В любой другой точке градиента значение цвета будет вычисляться методом интерполяции. Например:

```
// Линейный градиент, по диагонали холста (предполагается, что преобразования отсутствуют)
var bgfade = c.createLinearGradient(0,0,canvas.width,canvas.height);
bgfade.addColorStop(0.0, "#88f"); // От светло-синего вверху слева
bgfade.addColorStop(1.0, "#fff"); // До белого внизу справа

// Градиент между двумя концентрическими окружностями. Прозрачный в середине
// до полупрозрачного серого и опять до прозрачного.
var peekhole = c.createRadialGradient(300,300,100, 300,300,300);
peekhole.addColorStop(0.0, "transparent");         // Прозрачный
peekhole.addColorStop(0.7, "rgba(100,100,100,.9)"); // Полупрозрачный серый
peekhole.addColorStop(1.0, "rgba(0,0,0,0)");        // Опять прозрачный
```

Важно понимать, что градиенты тесно связаны с местоположением в холсте. При создании градиента вы указываете его границы. Если после этого попытаться выполнить заливку области за этими границами, будет использован сплошной цвет, соответствующий одному или другому концу градиента. Если, к примеру, вы определите градиент вдоль линии, соединяющей точки (0,0) и (100, 100), этот градиент можно будет использовать только для заливки объектов, находящихся внутри прямоугольной области (0,0,100,100).

Рисунок, изображенный на рис. 21.10, был создан с использованием шаблона `pattern` и градиентов `bgfade` и `peekhole` с помощью следующего программного кода:

```
c.fillStyle = bgfade;           // Сначала использовать линейный градиент
c.fillRect(0,0,600,600);        // Залить весь холст
c.strokeStyle = pattern;        // Использовать шаблон для рисования линий
c.lineWidth = 100;              // Очень толстые линии
c.strokeRect(100,100,400,400); // Нарисовать большой квадрат
```

```
c.fillStyle = peekhole;          // Использовать радиальный градиент
c.fillRect(0,0,600,600);         // Покрыть холст этой полупрозрачной заливкой
```

21.4.8. Атрибуты рисования линий

Вы уже знакомы со свойством `lineWidth`, которое определяет толщину линий, рисуемых методами `stroke()` и `strokeRect()`. Кроме свойства `lineWidth` (и конечно же, `strokeStyle`) существует еще три графических атрибута, влияющих на рисование линий.

По умолчанию свойство `lineWidth` имеет значение 1, и ему можно присвоить любое положительное целое число, и даже дробное число меньше 1. (Линии толщиной менее одного пиксела рисуются полупрозрачными цветами, поэтому они выглядят менее темными по сравнению с линиями толщиной в 1 пиксел). Чтобы полностью понять действие свойства `lineWidth`, представьте контур как комбинацию бесконечно тонких одномерных линий. Прямые линии и кривые, рисуемые методом `stroke()`, центрируются по этому контуру, выступая на половину `lineWidth` в обе стороны. Если при рисовании замкнутого контура необходимо, чтобы видимы были только части линий за пределами контура, нарисуйте сначала контур, а затем залейте его непрозрачным цветом, чтобы скрыть части линий, которые вторгаются внутрь контура. Или, если необходимо, чтобы видимы были только части линий внутри замкнутого контура, вызовите сначала методы `save()` и `clip()` (раздел 21.4.10), а затем методы `stroke()` и `restore()`.

Из-за имеющейся возможности изменять масштаб по осям координат, как показано на рис. 21.7, толщина линий зависит от текущего преобразования. Если выполнить вызов `scale(2,1)`, чтобы изменить масштаб по оси X и оставить нетронутым масштаб по оси Y, вертикальные линии будут получаться в два раза толще горизонтальных, нарисованных с одним и тем же значением свойства `lineWidth`. Важно понимать, что толщина линий определяется значением свойства `lineWidth` и текущим преобразованием, имевшимися на момент вызова метода `stroke()`, а не на момент вызова метода `lineTo()` или другого метода конструирования контура.

Три других атрибута рисования линий определяют внешний вид несоединенных концов контура и вершин, где соединяются два фрагмента контура. Они оказывают весьма несущественное влияние на внешний вид тонких линий, но обеспечивают существенные отличия при рисовании толстых линий. Действие двух этих свойств иллюстрируется изображением на рис. 21.11. Здесь контур показан как тонкая черная линия, а результат рисования линий – как окружающая ее серая область.

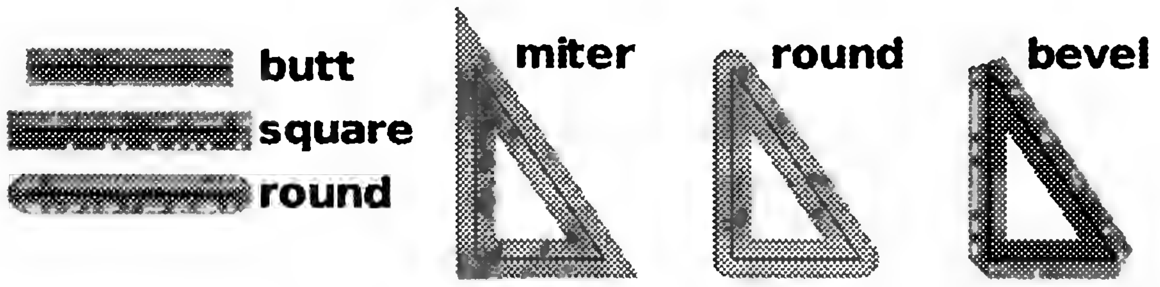


Рис. 21.11. Действие атрибутов lineCap и lineJoin

Свойство `lineCap` определяет, как будут выглядеть концы незамкнутых фрагментов контуров. Значение «butt» (по умолчанию) соответствует завершению линий непосредственно в конечной точке. При значении «square» линия будет продолжена за конечную точку на половину толщины и будет иметь квадратный конец. А при значении «round» линия будет продолжена за конечную точку на половину толщины и будет иметь закругленный конец (с радиусом закругления в половину толщины линии).

Свойство `lineJoin` определяет внешний вид вершин, соединяющих фрагменты контура. По умолчанию это свойство имеет значение «miter», при котором внешние края линий двух фрагментов контура будут продолжены, пока они не встретятся. При значении «round» вершины получают закругленную форму, а при значении «bevel» вершины обрезаются прямыми линиями.

Последнее свойство, связанное с рисованием линий, – это свойство `miterLimit`, которое используется, только когда свойство `lineJoin` имеет значение «miter». Когда две линии соединяются под острым углом, сопряжение между ними может оказаться довольно протяженным, и эти протяженные сопряжения могут нарушать визуальную гармонию. Свойство `miterLimit` определяет верхнюю границу протяженности сопряжений. Если сопряжение в некоторой вершине оказывается длиннее половины длины линии, умноженной на значение `miterLimit`, эта вершина будет нарисована с обрезанным сопряжением.

21.4.9. Текст

Для рисования текста в холсте обычно используется метод `fillText()`, который рисует текст, используя цвет (градиент или шаблон), определяемый свойством `fillStyle`. Если необходимо использовать специальные эффекты при выводе текста крупными символами, для рисования контуров отдельных символов можно применить метод `strokeText()` (пример вывода контуров символов приводится на рис. 21.13). Оба метода принимают в первом аргументе текст, который требуется нарисовать, и координаты X и Y вывода текста во втором и третьем аргументах. Ни один из этих методов не вносит изменений в текущий контур и не смещает текущую точку. Как видно на рис. 21.7, при выводе текста учитывается текущее преобразование системы координат.

Свойство `font` определяет шрифт, который будет использоваться для рисования текста. Значение этого свойства должно быть строкой с соблюдением синтаксиса CSS-атрибута `font`. Например:

```
"48pt sans-serif"
"bold 18px Times Roman"
"italic 12pt monospaced"
"bolder smaller serif" // жирнее и меньше, чем шрифт элемента <canvas>
```

Свойство `textAlign` определяет способ выравнивания текста по горизонтали с учетом координаты X, переданной методу `fillText()` или `strokeText()`. Свойство `textBaseline` определяет способ выравнивания текста по вертикали с учетом координаты Y. На рис. 21.12 показано действие всех допустимых значений этих свойств. Тонкая линия рядом с каждой строкой текста – это опорная линия шрифта, а маленький квадратик обозначает точку `(x,y)`, координаты которой были переданы методу `fillText()`.

По умолчанию свойство `textAlign` имеет значение «start». Обратите внимание, что для текста, который записывается слева направо, выравнивание, определяемое значением «start», совпадает с выравниванием, определяемым значением «left», аналогично совпадает и действие значений «end» «right». Однако если в элементе <canvas> определить атрибут `dir` со значением «rtl» (right-to-left – справа налево), действие значения «start» выравнивания будет совпадать с действием значения «right», а действие значения «end» – с действием значения «left».

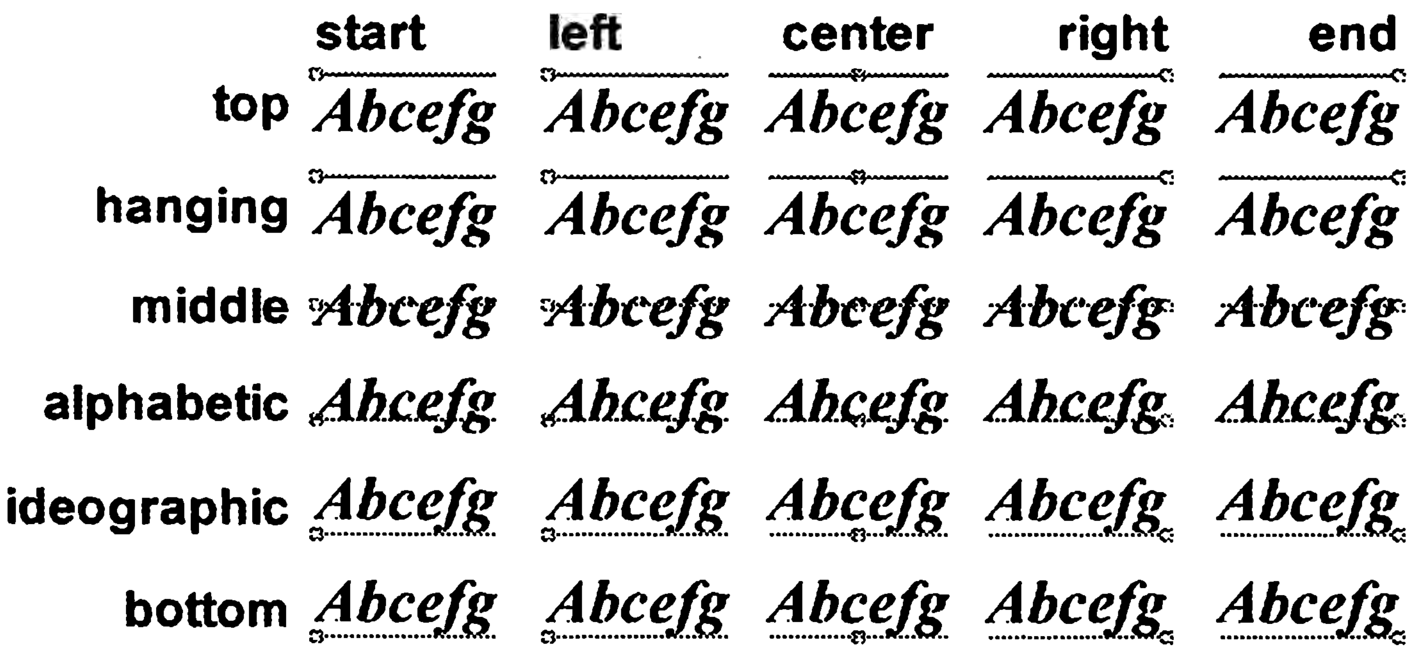


Рис. 21.12. Действие свойств textAlign и textBaseline

Свойство `textBaseline` по умолчанию имеет значение «alphabetic», которое соответствует алфавиту Latin и подобным ему. Значение «ideographic» используется совместно с идеографическими алфавитами, такими как китайский и японский. Значение «hanging» предназначено для использования со слоговыми и подобными им алфавитами (которые используются во многих языках в Индии). Значения «top», «middle» и «bottom» определяют исключительно геометрическое положение опорной линии шрифта, исходя из размеров «кегельной площадки» шрифта.

Методы `fillText()` и `strokeText()` принимают четвертый необязательный аргумент. Этот аргумент определяет максимальную ширину отображаемого текста. Если текст окажется шире указанного значения, при заданном значении свойства `font` будет выполнено его масштабирование или будет использован более узкий или более мелкий шрифт.

Если вам потребуется узнать размеры текста до его вывода, передайте его методу `measureText()`. Этот метод возвращает объект `TextMetrics`, определяющий размеры для текущего значения свойства `font`. На момент написания этих строк объект `TextMetrics` позволял определить только ширину текста. Определить ширину текстовой строки на экране можно следующим образом:

```
var width = c.measureText(text).width;
```

21.4.10. Отсечение

После определения контура обычно вызывается метод `stroke()` или `fill()` (или оба). Можно также вызвать метод `clip()`, чтобы определить область отсечения. После того как область отсечения будет определена, рисование будет выполняться только в ее пределах. На рис. 21.13 изображен сложный рисунок, полученный с использованием областей отсечения. Вертикальная полоса в середине и текст вдоль нижнего края рисунка были нарисованы до определения области отсечения, а заливка была выполнена после определения треугольной области отсечения.

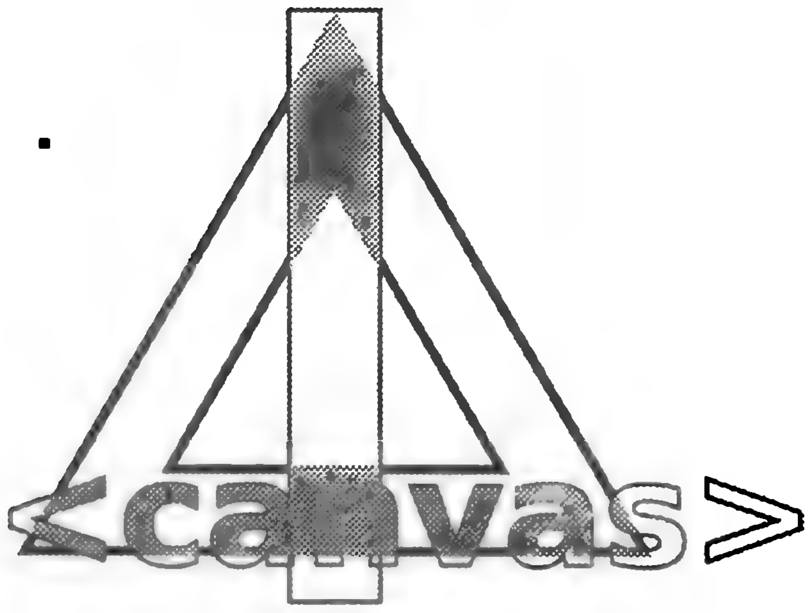


Рис. 21.13. Рисование контуров выполнено до, а заливка – после определения области отсечения

Изображение на рис. 21.13 было получено с помощью метода polygon() из примера 21.4 и следующего программного кода:

```
// Определить некоторые графические атрибуты
c.font = "bold 60pt sans-serif";   // Большой шрифт
c.lineWidth = 2;                   // Узкие
c.strokeStyle = "#000";            // и черные линии

// Контур прямоугольника и текст
c.strokeRect(175, 25, 50, 325);    // Вертикальная полоса в середине
c.strokeText("<canvas>", 15, 330); // strokeText() вместо fillText()

// Определить сложный контур, внутренняя область которого является внешней.
polygon(c,3,200,225,200);          // Большой треугольник
polygon(c,3,200,225,100,0,true);   // Нарисовать маленький треугольник
                                   // в обратном направлении

// Превратить этот контур в область отсечения.
c.clip();

// Нарисовать контур линиями толщиной 5 пикселов, внутри области отсечения.
c.lineWidth = 10; // Половина этой линии толщиной 10 пикселов окажется
                  // за пределами области отсечения
c.stroke();

// Залить область контура прямоугольника и текста, попавшую в область отсечения
c.fillStyle = "#aaa"             // Светло-серый
c.fillRect(175, 25, 50, 325);    // Залить вертикальную полосу
c.fillStyle = "#888"             // Темно-серый
c.fillText("<canvas>", 15, 330); // Залить текст
```

Важно отметить, что при вызове метода clip() выполняется усечение самого текущего контура, и этот новый усеченный контур становится новой областью отсечения. Это означает, что метод clip() может только сжимать область отсечения и не способен расширять ее. Метода, который позволил бы сбросить область отсечения, не существует, поэтому перед вызовом метода clip() следует вызвать метод save(), чтобы позднее можно было вернуться к неусеченному контуру вызовом метода restore().

21.4.11. Тени

Объект `CanvasRenderingContext2D` имеет четыре свойства графических атрибутов, управляющих рисованием теней. Если присвоить этим свойствам соответствующие значения, любые линии, области, текст или изображения будут отбрасывать тени, что создаст эффект расположения этих элементов над поверхностью холста. На рис. 21.14 показано, как выглядят тени, отбрасываемые закрашенным прямоугольником, контуром прямоугольника и закрашенным текстом.

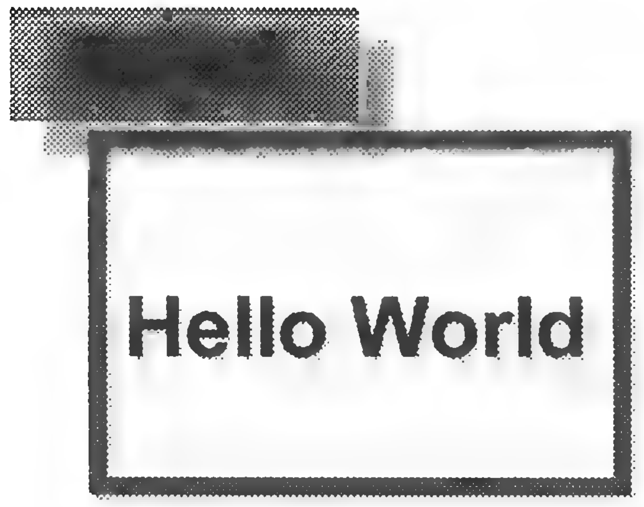


Рис. 21.14. Автоматически сгенерированные тени

Свойство `shadowColor` определяет цвет тени. Значением по умолчанию является полностью прозрачный черный цвет, и тени остаются невидимыми, если только не присвоить этому свойству значение, соответствующее полупрозрачному или непрозрачному цвету. Этому свойству допускается присваивать только строковые значения цвета: для рисования теней не могут использоваться шаблоны и градиенты. Использование полупрозрачных цветов дает более реалистичное изображение теней из-за просвечивания фона.

Свойства `shadowOffsetX` и `shadowOffsetY` определяют смещение тени по осям X и Y. По умолчанию оба свойства имеют значение 0, что соответствует размещению тени непосредственно под рисунком, где она невидима. Если присвоить обоим свойствам положительные значения, тени будут нарисованы правее и ниже рисунка, как если бы источник света, освещающий холст, находился за пределами экрана левее и выше. Чем больше смещение, тем длиннее отбрасываемая тень и тем «выше» над холстом будет казаться нарисованный объект.

Свойство `shadowBlur` определяет, насколько размытым будет выглядеть край тени. По умолчанию это свойство имеет значение 0, которому соответствую четкие, неразмытые тени. Большие значения производят более сильный эффект размытия, до определенной степени, определяемой реализацией. Значение этого свойства используется как параметр Гауссовой функции размытия и не является размером или длиной в пикселах.

В примере 21.8 представлен программный код, который использовался для получения изображения на рис. 21.14 и демонстрирующий использование всех четырех свойств, управляющих отображением теней.

Пример 21.8. Установка параметров тени

```
// Определить узкую тень
c.shadowColor = "rgba(100,100,100,.4)"; // Полупрозрачный серый цвет
c.shadowOffsetX = c.shadowOffsetY = 3;  // Тень смещена вправо вниз
```

```
c.shadowBlur = 5;                     // Размытые границы тени

// Нарисовать текст в синем прямоугольнике с этими параметрами тени
c.lineWidth = 10;
c.strokeStyle = "blue";
c.strokeRect(100, 100, 300, 200);     // Нарисовать прямоугольник
c.font = "Bold 36pt Helvetica";
c.fillText("Hello World", 115, 225);  // Нарисовать текст

// Определить более широкую тень. Большие значения смещений создают эффект более высокого
// расположения объекта. Обратите внимание, как полупрозрачная тень смешивается
// с синим контуром прямоугольника.
c.shadowOffsetX = c.shadowOffsetY = 20;
c.shadowBlur = 10;
c.fillStyle = "red";               // Нарисовать сплошной красный прямоугольник,
c.fillRect(50,25,200,65);          // располагающийся выше синего прямоугольника
```

Значения свойств `shadowOffsetX` и `shadowOffsetY` всегда определяются в системе координат по умолчанию и не подвержены действию методов `rotate()` и `scale()`. Допустим, к примеру, что вы повернули систему координат на 90 градусов, чтобы нарисовать текст по вертикали и затем вернулись к прежней системе координат, чтобы нарисовать текст по горизонтали. Обе текстовые надписи, вертикальная и горизонтальная, будут отбрасывать тень в одном направлении, что обычно соответствует нашим представлениям. Аналогично фигуры, нарисованные с применением различных преобразований, будут иметь тени с одинаковой «высотой».[1]

21.4.12. Изображения

Помимо векторной графики (контуры, линии и прочее), прикладной интерфейс объекта `Canvas` поддерживает также растровые изображения. Метод `drawImage()` копирует в холст пикселы из исходного изображения (или из прямоугольной области исходного изображения), выполняя операции масштабирования и вращения, если они необходимы.

Метод `drawImage()` может вызываться с тремя, пятью или девятью аргументами. Во всех случаях в первом аргументе ему передается исходное изображение. Часто в этом аргументе передается элемент `<img>` или неотображаемый объект `Image`, созданный с помощью конструктора `Image()`. Однако точно так же в первом аргументе можно передать другой элемент `<canvas>` или даже элемент `<video>`. Если методу `drawImage()` передать элемент `<img>` или `<video>`, который к этому моменту еще не завершил загрузку изображения, он ничего не скопирует.

При вызове с тремя аргументами во втором и третьем аргументах методу `drawImage()` передаются координаты X и Y верхнего левого угла области, в которую должно быть скопировано изображение. В этом случае в холст будет скопировано изображение целиком. Координаты X и Y будут интерпретироваться как координаты в текущей системе координат, поэтому при необходимости изображение будет масштабировано или повернуто.

При вызове метода `drawImage()` с пятью аргументами к аргументам с координатами X и Y, описанным выше, добавляются ширина и высота. Эти четыре аргумен-

---

[1] На момент написания этих строк в версии 5 броузера Google Chrome тени были реализованы с ошибкой, и их смещения были подвержены действию преобразований.

та определяют прямоугольную область внутри холста. Верхний левый угол исходного изображения будет помещен в точку (x,y), а правый нижний – в точку (x+width, y+height). Опять же в холст будет скопировано изображение целиком. Прямоугольная область назначения определяется в текущей системе координат. Эта версия метода выполнит масштабирование изображение, чтобы уместить его в отведенную область, даже если к исходной системе координат не применялось преобразование масштабирования.

При вызове метода drawImage() с девятью аргументами ему передаются координаты и размеры области в исходном изображении и области в холсте, и он скопирует только указанную область исходного изображения. В аргументах со второго по пятый указываются координаты и размеры прямоугольной области в исходном изображении. Они измеряются в пикселах CSS. Если исходное изображение представлено другим элементом <canvas>, координаты и размеры исходного изображения будут измеряться в системе координат по умолчанию и никакие преобразования, применявшиеся к системе координат исходного холста, учитываться не будут. В аргументах с шестого по девятый указываются координаты и размеры области в текущей (а не по умолчанию) системе координат, куда будет скопирован указанный фрагмент изображения.

Пример 21.9 демонстрирует простой случай применения метода drawImage(). В нем используется версия метода с девятью аргументами, чтобы скопировать фрагмент холста, увеличенный и повернутый, обратно в тот же самый холст. Как видно на рис. 21.15, изображение было увеличено достаточно, чтобы проявилась его растровая структура и можно было наблюдать полупрозрачные пикселы, которые использованы для сглаживания краев линии.

Рис. 21.15. Копия изображения была увеличена методом drawImage()

Пример 21.9. Использование метода drawImage()

```
// Нарисовать линию в верхнем левом углу
c.moveTo(5,5);
c.lineTo(45,45);
c.lineWidth = 8;
c.lineCap = "round";
c.stroke();

// Определить преобразование системы координат
c.translate(50,100);
c.rotate(-45*Math.PI/180); // Разгладить линию
c.scale(10,10);            // Увеличить ее, чтобы были видны отдельные пикселы

// С помощью drawImage скопировать линию
c.drawImage(c.canvas,
            0, 0, 50, 50,  // исходная область: непреобразованная
            0, 0, 50, 50); // область назначения: преобразованная
```

Помимо возможности копировать изображение в холст, имеется также возможность извлекать содержимое холста в виде изображения с помощью метода `toDataURL()`. В отличие от других методов, описанных выше, метод `toDataURL()` – это метод самого элемента `Canvas`, а не объекта `CanvasRenderingContext2D`. Обычно метод `toDataURL()` вызывается без аргументов и возвращает содержимое холста как PNG-изображение, закодированное в виде строки в формате URL `data:`. Возвращаемая строка URL подходит для использования в элементе `<img>`, благодаря чему можно создать статический снимок холста, как показано ниже:

```
var img = document.createElement("img"); // Создать элемент <img>
img.src = canvas.toDataURL();            // Установить его атрибут src
document.body.appendChild(img);          // Добавить элемент в документ
```

Все броузеры в обязательном порядке поддерживают формат PNG изображений. Некоторые реализации могут также поддерживать другие форматы, и вы можете указать желаемый MIME-тип в необязательном первом аргументе в вызове метода `toDataURL()`. Подробности смотрите в справочном разделе книги.

Существует одно важное ограничение, связанное с безопасностью, о котором следует знать, планируя применять метод `toDataURL()`. Чтобы предотвратить утечку информации между доменами, метод `toDataURL()` не работает с элементами `<canvas>`, имеющими «неясное происхождение». Считается, что холст имеет неясное происхождение, если в него вставлялось изображение (непосредственно, вызовом метода `drawImage()`, или косвенно, с помощью метода `CanvasPattern`), имеющее происхождение, отличное от происхождения документа, содержащего элемент `<canvas>`.

21.4.13. Композиция

При рисовании линий и текста, заливке областей или копировании изображений может получиться так, что новые пикселы будут накладываться сверху на уже существующие в холсте. При рисовании непрозрачных пикселов они просто будут замещать уже имеющиеся пикселы. Но при рисовании полупрозрачных пикселов новые («исходные») пикселы будут объединяться (комбинироваться) со старыми («целевыми») пикселами так, что старые пикселы будут видны сквозь новые, с учетом степени прозрачности этих пикселов.

Этот процесс объединения новых полупрозрачных исходных пикселов с существующими целевыми пикселами называется *композицией*, а процесс композиции, описанный выше, используется по умолчанию при объединении пикселов. Однако композиция нужна не всегда. Представьте, что вы нарисовали в холсте рисунок, использовав полупрозрачный цвет, и теперь хотите внести в холст временные изменения, а позднее восстановить оригинальный рисунок. Самый простой способ реализовать это состоит в том, чтобы холст (или его область) скопировать в другой, неотображаемый холст с помощью метода `drawImage()`. А затем, когда придет время восстановить холст, можно скопировать обратно в него фрагмент, который был сохранен в неотображаемом холсте. Напомню, что скопированные пикселы были полупрозрачными. Если режим композиции будет действовать, они не затрут полностью временные изменения. В подобных ситуациях необходимо иметь способ отключать композицию: чтобы вставлять исходные пикселы, игнорируя целевые, независимо от степени прозрачности исходных.

Установить тип композиции можно с помощью свойства `globalCompositeOperation`. По умолчанию оно имеет значение «source-over», в соответствии с которым исходные пикселы накладываются «поверх» («over») целевых пикселов и объединяются с ними, если исходные пикселы являются полупрозрачными. Если присвоить этому свойству значение «copy», композиция будет отключена: исходные пикселы будут скопированы в холст без изменений и затрут целевые пикселы. Иногда может оказаться полезным еще одно значение свойства `globalCompositeOperation` – «destination-over». При использовании этого вида композиции пикселы объединяются так, как будто исходные пикселы добавляются под существующие целевые пикселы. Если целевые пикселы будут иметь прозрачный или полупрозрачный цвет, некоторые или все исходные пикселы будут видны сквозь этот цвет.

«source-over», «destination-over» и «copy» – это три наиболее часто используемых вида композиции, однако прикладной интерфейс объекта `Canvas` поддерживает 11 значений для атрибута `globalCompositeOperation`. Названия этих видов композиции достаточно красноречиво объясняют, что они призваны делать, и вам может потребоваться немало времени, чтобы разобраться в особенностях разных видов композиции, подставляя их названия в примеры, демонстрирующие их действие.

На рис. 21.16 показаны все 11 видов композиции пикселов с «предельными» значениями прозрачности: все пикселы являются полностью непрозрачными или полностью прозрачными. В каждом из 11 прямоугольников сначала рисовался квадрат, который представляет целевые пикселы. Затем устанавливалось свойство `globalCompositeOperation` и рисовался круг, представляющий исходные пикселы.

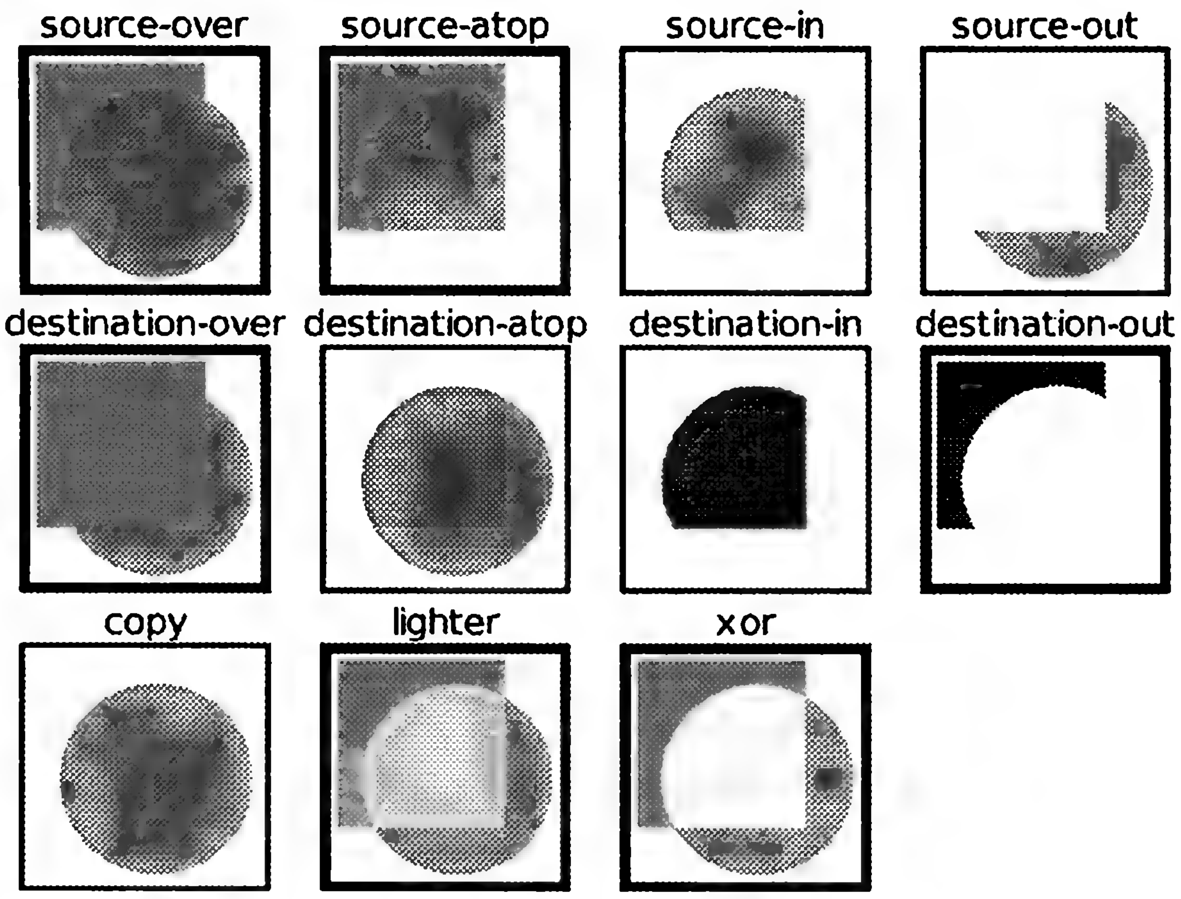


Рис. 21.16. Виды композиции пикселов с предельными значениями прозрачности

На рис. 21.17 изображен похожий пример, в котором использовались пикселы с «промежуточными» значениями прозрачности. В этой версии исходный круг и целевой квадрат были залиты градиентами, вследствие чего в рисунке присутствуют пикселы с различной степенью прозрачности.

Вы можете обнаружить, что понять действие того или иного вида композиции совсем непросто, когда в операцию вовлечены полупрозрачные пикселы, как в данном примере. Если у вас есть желание более глубоко разобраться с композицией, в справочной статье `CanvasRenderingContext2D` вы найдете уравнения вычисления конечного значения цвета пиксела, исходя из значений цвета исходного и целевого пикселов для всех 11 видов композиции.

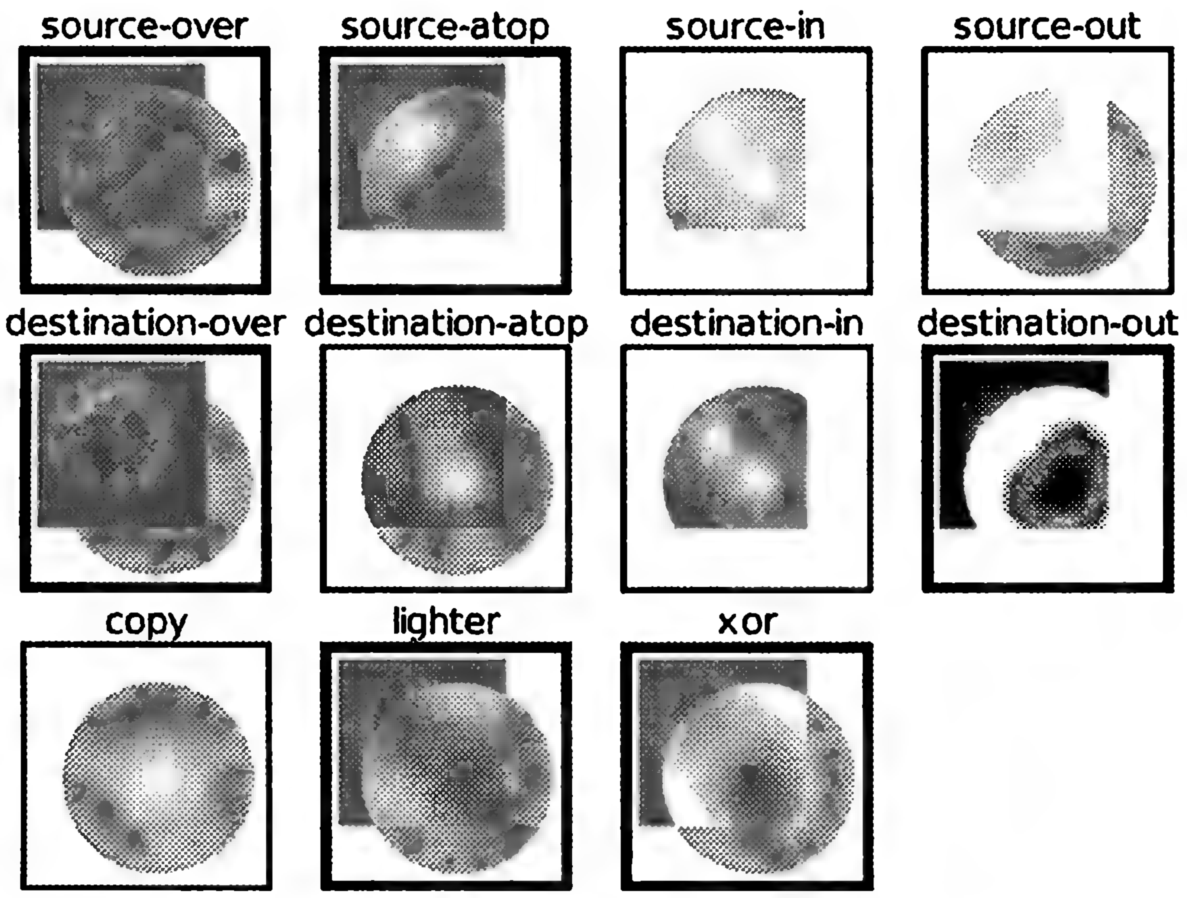


Рис. 21.17. Виды композиции пикселов с промежуточными значениями прозрачности

На момент написания этих строк производители броузеров не смогли прийти к единству в реализации 5 из 11 видов композиции: «copy», «source-in», «source-out», «destination-atop» и «destination-in» действуют по-разному в разных броузерах и не могут использоваться переносимым образом. Далее приводится подробное описание этих различий, но если вы не планируете использовать эти виды композиций, то можете сразу перейти к следующему разделу.

При использовании пяти видов композиции, перечисленных выше, цвет целевых пикселов либо игнорируется при вычислении результатов, либо получающиеся пикселы делаются прозрачными, если прозрачными являются исходные пикселы. Различия в реализациях связаны с определением исходных пикселов. Броузеры Safari и Chrome выполняют композицию «локально»: в расчетах участвуют только фактические исходные пикселы, которые выводятся методами `fill()`, `stroke()` и другими. IE9, вероятно, последует этому примеру. Броузеры Firefox и Opera выполняют композицию «глобально»: в расчетах участвуют все пикселы в текущей области отсечения при выполнении любой операции рисования. Если для данного целевого пиксела отсутствует исходный пиксел, считается, что исходный пиксел имеет черный прозрачный цвет. В Firefox и Opera это означает, что пять видов композиции, перечисленные выше, фактически стирают целевые пикселы внутри области отсечения там, где отсутствуют исходные пикселы. Изображения на рис. 21.16 и рис. 21.17 были получены в броузере Firefox. Это объясняет, почему рамки, окружающие примеры применения видов композиции «copy», «source-in», «source-out», «destination-atop» и «destination-in», тоньше, чем рамки вокруг других примеров: прямоугольник, окружающий каждый пример, является областью отсечения, и применение этих пяти видов композиции приводит к стиранию части рамки (половина `lineWidth`), попадающей внутрь контура. Для сравнения на рис. 21.18 показаны те же результаты, что и на рис. 21.17, но полученные в броузере Chrome.

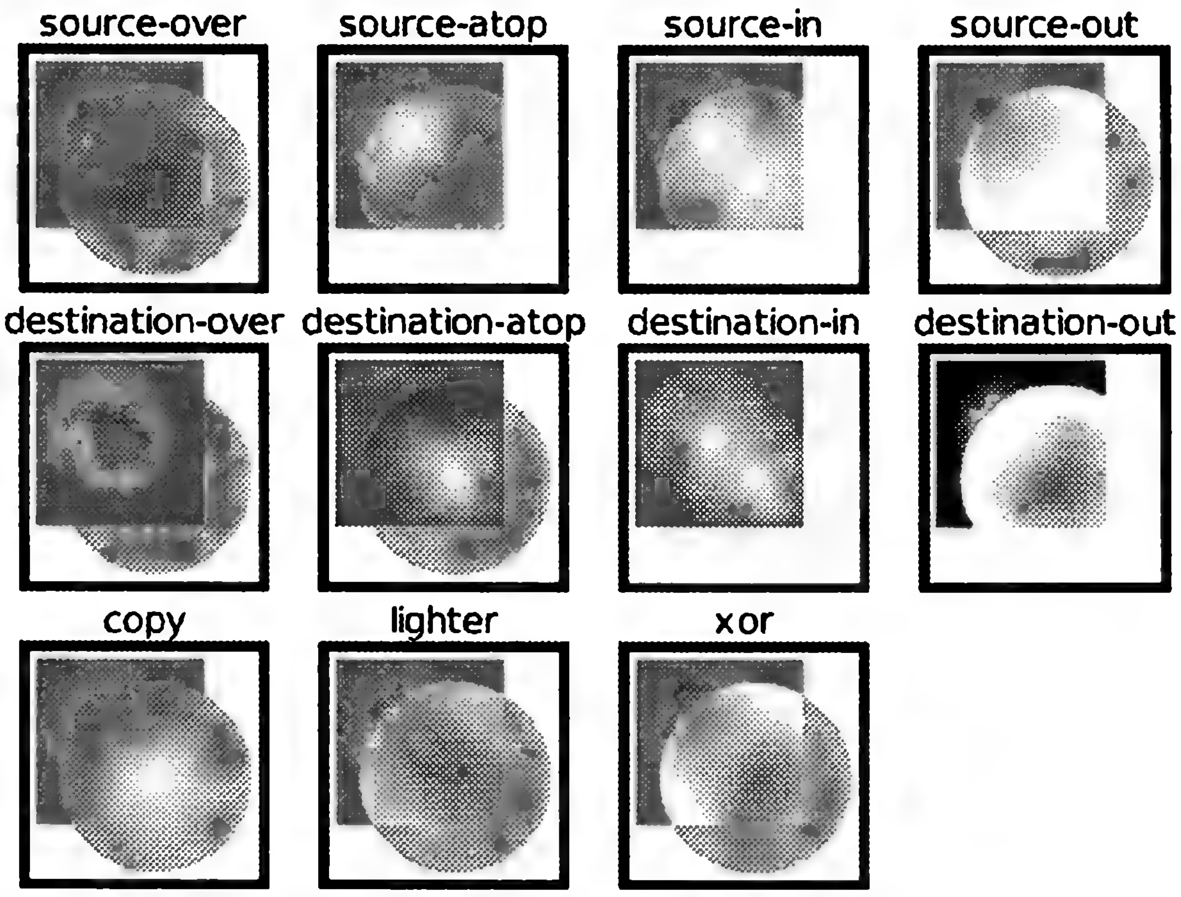


Рис. 21.18. Локальная композиция

На момент написания этих строк проект стандарта HTML5 утвердил глобальный подход к композиции, реализованный в броузерах Firefox и Opera. Производители броузеров знают об имеющейся несовместимости и не удовлетворены текущим состоянием спецификации. Велика вероятность, что спецификация будет пересмотрена в пользу локальной композиции.

Наконец, обратите внимание, что в броузерах, таких как Safari и Chrome, реализующих локальную композицию, существует возможность реализовать глобальную композицию. Сначала нужно создать пустой неотображаемый холст с теми же размерами, что и отображаемый. Затем нарисовать исходное изображение на неотображаемом холсте и вызвать метод `drawImage()`, чтобы скопировать неотображаемый рисунок в отображаемый холст и применить глобальную композицию в пределах области отсечения. В броузерах, таких как Firefox, реализующих глобальную композицию, нет универсального приема выполнения локальной композиции, но нередко можно получить достаточно близкий эффект, определив соответствующую область отсечения перед выполнением операции рисования, для которой композиция должна выполняться локально.

21.4.14. Манипулирование пикселами

Метод `getImageData()` возвращает объект `ImageData`, представляющий массив пикселов (в виде компонентов R, G, B и A) из прямоугольной области холста. Создать пустой объект `ImageData` можно с помощью метода `createImageData()`. Пикселы в объекте `ImageData` доступны для записи, благодаря чему их можно изменить как угодно и затем скопировать обратно в холст вызовом метода `putImageData()`.

Эти методы манипулирования пикселами предоставляют низкоуровневый доступ к холсту. Координаты и размеры области, передаваемой методу `getImageData()`, задаются в системе координат по умолчанию: ее размеры измеряются в CSS-пикселах и без учета текущего преобразования. При вызове метода `putImageData()` координаты также задаются в системе координат по умолчанию. Кроме того, метод `putImageData()` игнорирует все графические атрибуты. Он не использует механизм композиции, не умножает пикселы на значение свойства `globalAlpha` и не рисует тени.

Методы манипулирования пикселами могут пригодиться для реализации обработки изображений. Пример 21.10 демонстрирует, как создать простейший эффект размытия или «смазывания» быстро движущегося объекта в элементе <canvas>. Пример демонстрирует применение методов getImageData() и putImageData() и показывает, как выполнять итерации по пикселам в объекте ImageData и изменять их значения, но без подробного описания. Полная информация о методах getImageData() и putImageData() приводится в справочной статье CanvasRenderingContext2D, а подробное описание объекта ImageData – в его собственной справочной статье.

Пример 21.10. Создание эффекта размытия быстродвижущегося объекта с помощью объекта ImageData

```
// "Смазать" пикселы прямоугольной области вправо, чтобы воспроизвести эффект быстрого
// движения объекта справа налево. Значение n должно быть равно или больше 2. Чем больше
// значение, тем сильнее эффект смазывания. Координаты и размеры прямоугольной области
// задаются в системе координат по умолчанию.
function smear(c, n, x, y, w, h) {
    // Получить объект ImageData, представляющий пикселы области эффекта
    var pixels = c.getImageData(x,y,w,h);

    // Смазывание выполняется на месте, и потому требуется получить только
    // исходный объект ImageData. Некоторые алгоритмы обработки изображений требуют
    // использования дополнительного объекта ImageData для сохранения трансформированных
    // значений пикселов. Если бы потребовался промежуточный буфер вывода, можно было бы
    // создать новый объект ImageData с теми же размерами следующим способом:
    //   var output_pixels = c.createImageData(pixels);

    // Эти размеры могут отличаться от значений аргументов w и h: на каждый
    // CSS-пиксел может приходиться несколько аппаратных пикселов.
    var width = pixels.width, height = pixels.height;

    // Это массив байтов, хранящий информацию о пикселах, слева направо и сверху вниз.
    // Для каждого пиксела отводится 4 последовательных байта, в порядке R,G,B,A.
    var data = pixels.data;

    // Смазать каждый пиксел после первого в каждой строке, заменив его суммой
    // 1/n-й доли его собственного значения и m/n-й доли значения предыдущего пиксела
    var m = n-1;
    for(var row = 0; row < height; row++) {            // Для каждой строки
        var i = row*width*4 + 4;                      // Индекс второго пиксела в строке
        for(var col = 1; col < width; col++, i += 4) { // Для каждого столбца
            data[i] = (data[i] + data[i-4]*m)/n;       // Красная составляющая
            data[i+1] = (data[i+1] + data[i-3]*m)/n;   // Зеленая
            data[i+2] = (data[i+2] + data[i-2]*m)/n;   // Синяя
            data[i+3] = (data[i+3] + data[i-1]*m)/n;   // Альфа-составляющая
        }
    }

    // Скопировать смазанное изображение обратно в ту же позицию в холсте
    c.putImageData(pixels, x, y);
}
```

Обратите внимание, что на метод getImageData() накладываются те же ограничения политики общего происхождения, что и на метод toDataURL(): он не будет работать с холстами, в которые вставлялись изображения (непосредственно, вызовом метода drawImage(), или косвенно, с помощью метода CanvasPattern), имеющие происхождение, отличное от происхождения документа, содержащего элемент <canvas>.

21.4.15. Определение попадания

Метод `isPointInPath()` позволяет узнать, находится ли указанная точка внутри (или на границе) текущего контура, и возвращает `true`, если это так, и `false` – в противном случае. Метод принимает координаты точки в не преобразованной системе координат по умолчанию. Это позволяет использовать метод для *определения попадания*: определения принадлежности точки, где был выполнен щелчок мышью, некоторой определенной фигуре.

Однако значения свойств `clientX` и `clientY` объекта `MouseEvent` нельзя передать непосредственно методу `isPointInPath()`. Во-первых, координаты события мыши следует преобразовать из координат объекта `Window` в относительные координаты элемента `<canvas>`. Во-вторых, если экранные размеры холста отличаются от его фактических размеров, координаты мыши необходимо перевести в соответствующий масштаб. В примере 21.11 показана вспомогательная функция, используемая для определения попадания точки события `MouseEvent` в текущий контур.

Пример 21.11. Проверка попадания точки события мыши в текущий контур

```
// Возвращает true, если указанное событие мыши возникло в текущем контуре
// в указанном объекте CanvasRenderingContext2D.
function hitpath(context, event) {
    // Получить элемент <canvas> из объекта контекста
    var canvas = context.canvas;

    // Получить координаты и размеры холста
    var bb = canvas.getBoundingClientRect();

    // Преобразовать и масштабировать координаты события мыши в координаты холста
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Вызвать isPointInPath с преобразованными координатами
    return context.isPointInPath(x,y);
}
```

Эту функцию `hitpath()` можно использовать в обработчиках событий, как показано ниже:

```
canvas.onclick = function(event) {
    if (hitpath(this.getContext("2d"), event) {
        alert("Есть попадание!"); // Щелчок в пределах текущего контура
    }
};
```

Вместо проверки попадания в текущий контур с помощью метода `getImageData()` можно определить, окрашен ли пиксел в точке события мыши. Если пиксел (или пикселы) под указателем мыши оказался полностью прозрачным, следовательно, под указателем мыши ничего не было нарисовано и щелчок был выполнен за пределами какой-либо фигуры. Пример 21.12 демонстрирует, как реализовать подобную проверку попадания.

Пример 21.12. Проверка наличия окрашенного пиксела в точке события мыши

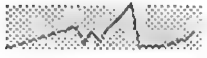
```
// Возвращает true, если указанное событие мыши возникло в точке,
// где находится непрозрачный пиксел.
function hitpaint(context, event) {
    // Преобразовать и масштабировать координаты события мыши в координаты холста
    var canvas = context.canvas;
    var bb = canvas.getBoundingClientRect();
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Получить пиксел (или пикселы, если одному CSS-пикселу соответствует
    // несколько аппаратных пикселов)
    var pixels = c.getImageData(x,y,1,1);

    // Если хотя бы один пиксел имеет ненулевое значение альфа-канала,
    // вернуть true (попадание)
    for(var i = 3; i < pixels.data.length; i+=4) {
        if (pixels.data[i] !== 0) return true;
    }

    // Иначе это был промах.
    return false;
}
```

21.4.16. Пример использования элемента <canvas>: внутристрочные диаграммы

Закончим главу практическим примером рисования внутристрочных диаграмм. *Внутристрочная диаграмма* (*sparkline*) – это маленькое изображение (обычно некий график) предназначенное для отображения в потоке текста, например: Server load:  8. Термин «sparkline» был введен их автором Эдвардом Тафти (Edward Tufte), который описывает внутристрочные диаграммы так: «Маленькие графические изображения с высоким разрешением, встроенные в контекст окружающих их слов, чисел, изображений. Внутристрочная диаграмма – это простой в создании, тесно связанный с данными график, размер которого сопоставим с размером слова». (Подробнее о создании внутристрочных диаграмм см. в книге Эдварда Тафти «Beautiful Evidence» [Graphics Press].)

В примере 21.13 приводится относительно простой модуль на языке JavaScript, позволяющий вставлять в веб-страницы внутристрочные диаграммы. Порядок работы модуля описывается в комментариях. Обратите внимание, что в нем используется функция `onLoad()` **из примера 13.5.**

Пример 21.13. Реализация внутристрочных диаграмм с помощью элемента <canvas>

```
/*
 * Отыскивает все элементы с CSS-классом "sparkline", анализирует их содержимое
 * как последовательность чисел и замещает их графическим представлением.
 *
 * Определить внутристрочную диаграмму в разметке можно так:
 *   <span class="sparkline">3 5 7 6 6 9 11 15</span>
 *
 * Определить визуальное оформление диаграмм средствами CSS можно так:
 *   .sparkline { background-color: #ddd; color: red; }
 *
 * - Цвет кривой графика определяется CSS-свойством color вычисленного стиля.
 * - Сами диаграммы являются прозрачными, поэтому сквозь них просвечивает фон страницы.
 * - Высота диаграммы определяется атрибутом data-height, если он указан,
 *   или свойством font-size вычисленного стиля в противном случае.
 * - Ширина диаграммы определяется атрибутом data-width, если он указан,
 *   или числом точек данных, умноженным на значение атрибута data-dx, если он
 *   указан, или числом точек данных, умноженным на высоту, деленную на 6
 * - Минимальное и максимальное значение по оси y извлекаются из атрибутов
 *   data-ymin и data-ymax, если они указаны, иначе отыскиваются минимальное
 *   и максимальное значение в данных.
 */
onLoad(function() { // Когда документ будет загружен
    //Отыскать все элементы с классом "sparkline"
    var elts = document.getElementsByClassName("sparkline");
    main: for(var e = 0; e < elts.length; e++) { // Для каждого элемента
        var elt = elts[e];

        // Получить содержимое элемента и преобразовать его в массив чисел.
        // Если преобразование не удалось, пропустить этот элемент.
        var content = elt.textContent || elt.innerText;  // Содержимое
        var content = content.replace(/^\s+|\s+$/g, ""); // Удалить пробелы
        var text = content.replace(/#.*$/gm, ""); // Удалить комментарии
        text = text.replace(/[\n\r\t\v\f]/g, " ");// Преобр. \n и др. в пробел
        var data = text.split(/\s+|\s*,\s*/);     // По пробелам и запятым
        for(var i = 0; i < data.length; i++) {    // Каждый фрагмент
            data[i] = Number(data[i]);            // Преобразовать в число
            if (isNaN(data[i])) continue main;    // Прервать при неудаче
        }

        // Определить цвет, ширину, высоту и границы по оси y для диаграммы
        // из данных, из атрибутов data- элемента и из вычисленного стиля.
        var style = getComputedStyle(elt, null);
        var color = style.color;
        var height = parseInt(elt.getAttribute("data-height")) ||
            parseInt(style.fontSize) || 20;
        var width = parseInt(elt.getAttribute("data-width")) ||
            data.length * (parseInt(elt.getAttribute("data-dx")) || height/6);
        var ymin = parseInt(elt.getAttribute("data-ymin")) || Math.min.apply(Math, data);
        var ymax = parseInt(elt.getAttribute("data-ymax")) || Math.max.apply(Math, data);
        if (ymin >= ymax) ymax = ymin + 1;

        // Создать элемент <canvas>.
        var canvas = document.createElement("canvas");
        canvas.width = width;    // Установить размеры холста
        canvas.height = height;
        canvas.title = content;  // Содержимое использовать как подсказку
        elt.innerHTML = "";      // Стереть содержимое элемента
        elt.appendChild(canvas); // Вставить холст в элемент

        // Нарисовать график по точкам (i,data[i]), преобразовав в координаты холста.
        var context = canvas.getContext('2d');
        for(var i = 0; i < data.length; i++) {         // Для каждой точки на графике
            var x = width*i/data.length;               // Масштабировать i
            var y = (ymax-data[i])*height/(ymax-ymin); // и data[i]
            context.lineTo(x,y); // Первый вызов lineTo() выполнит moveTo()
        }
        context.strokeStyle = color; // Указать цвет кривой на диаграмме
        context.stroke();            // и нарисовать ее
    }
});
```

22

Прикладные интерфейсы HTML5

Под термином HTML5 обычно подразумевается последняя версия спецификации языка разметки HTML, но этот термин также используется для обозначения целого комплекса веб-технологий, которые разрабатываются и определяются как часть языка разметки HTML или сопутствующие ему. Официально этот комплекс технологий называется «Open Web Platform». Однако на практике чаще используется сокращенное название «HTML5», и в данной главе мы будем использовать его именно в этом смысле. В других главах этой книги уже описывались некоторые новые прикладные интерфейсы HTML5:

- В главе 15 были представлены методы `getElementsByClassName()`, `querySelectorAll()` и атрибуты данных элементов документа.
- В главе 16 было описано свойство `classList` элементов.
- В главе 18 рассказывалось о спецификации «XMLHttpRequest Level 2», о выполнении междоменных HTTP-запросов и о прикладном интерфейсе `EventSource`, определяемом спецификацией «Server-Sent Events».
- В главе 20 был описан прикладной интерфейс Web Storage и кэш приложений для автономных веб-приложений.
- В главе 21 были представлены элементы `<audio>`, `<video>` и `<canvas>`, а также средства для работы с векторной графикой SVG.

Эта глава охватывает ряд других прикладных интерфейсов HTML5:

- В разделе 22.1 рассматривается прикладной интерфейс объекта `Geolocation`, позволяющий броузерам определять географическое местонахождение пользователя (с его разрешения).
- В разделе 22.2 рассказывается о прикладных интерфейсах управления историей посещений, которые позволяют веб-приложениям сохранять и обновлять информацию о своем состоянии в ответ на использование кнопок Back (Назад) и Forward (Вперед) броузера, без необходимости выполнять полную перезагрузку страницы с веб-сервера.
- В разделе 22.3 описывается простой прикладной интерфейс передачи сообщений между документами с различным происхождением. Он обеспечивает без-

опасный способ обхода ограничений политики общего происхождения (раздел 13.6.2), препятствующих непосредственному взаимодействию документов, полученных от разных веб-серверов.

- В разделе 22.4 охватывается новая важная особенность HTML5: возможность выполнять программный код на языке JavaScript в отдельном фоновом потоке выполнения и безопасно взаимодействовать с этими «рабочими» потоками.
- В разделе 22.5 рассматриваются специальные типы данных для работы с массивами байтов и чисел, более экономно расходующие память.
- В разделе 22.6 будут представлены двоичные объекты (`Blob`): нетипизированные порции данных, которые служат основным форматом обмена данными, используемым различными новыми прикладными интерфейсами для работы с двоичными данными. В этом разделе также охватывается несколько типов данных, связанных с большими двоичными объектами, и их прикладные интерфейсы: объекты `File` и `FileReader`, тип `BlobBuilder` и URL-адреса вида `blob://`.
- В разделе 22.7 демонстрируется прикладной интерфейс к файловой системе, посредством которого веб-приложения могут читать и писать файлы в частной файловой системе. Это один из пока не устоявшихся прикладных интерфейсов, и он не описывается в справочном разделе книги.
- В разделе 22.8 демонстрируется прикладной интерфейс объекта `IndexedDB`, предназначенный для сохранения и извлечения объектов в простых базах данных. Как и интерфейс к файловой системе, прикладной интерфейс IndexedDB пока является нестабильным и не описывается в справочном разделе.
- Наконец, раздел 22.9 охватывает прикладной интерфейс веб-сокетов Web Sockets, позволяющий веб-приложениям устанавливать соединения с веб-серверами и использовать двунаправленные сетевые каналы на основе потоков вместо модели сетевых взаимодействий типа запрос/ответ, поддерживаемой с помощью объекта `XMLHttpRequest`.

Особенности, описываемые в этой главе, либо не укладываются естественным образом ни в одну из тем, обсуждавшихся в предыдущих главах, либо пока не являются достаточно стабильными и зрелыми, чтобы обсуждать их в основных главах этой книги. Некоторые из прикладных интерфейсов выглядят достаточно стабильными, чтобы их можно было описать в справочном разделе, тогда как другие все еще продолжают изменяться и потому не были включены в четвертую часть книги. На тот момент, когда книга была отправлена в печать, все примеры в этой главе, кроме одного (пример 22.9), работали, по крайней мере, в одном из броузеров. Поскольку спецификации, описываемые здесь, все еще продолжают дорабатываться, некоторые из этих примеров могут перестать работать, когда вы будете читать эту главу.

22.1. Геопозиционирование

Прикладной интерфейс объекта `Geolocation` (*http://www.w3.org/TR/geolocation-API/*) позволяет программам на языке JavaScript запрашивать у броузера географическое местонахождение пользователя. Такие приложения могут отображать карты, маршруты и другую информацию, связанную с текущим местонахождением пользователя. При этом, конечно, возникает важная проблема соблюдения тайны частной информации, поэтому броузеры, поддерживающие прикладной интерфейс Geolocation, всегда запрашивают у пользователя подтверждение, прежде чем передать JavaScript-программе информацию о физическом местонахождении пользователя.

Броузеры с поддержкой интерфейса Geolocation определяют свойство navigator.geolocation. Это свойство ссылается на объект с тремя методами:

navigator.geolocation.getCurrentPosition()

Запрашивает текущее географическое местонахождение пользователя.

navigator.geolocation.watchPosition()

Не только запрашивает текущее местонахождение, но и продолжает следить за координатами, вызывая указанную функцию обратного вызова при изменении местонахождения пользователя.

navigator.geolocation.clearWatch()

Останавливает слежение за местонахождением пользователя. В аргументе этому методу следует передавать число, возвращаемое соответствующим вызовом метода watchPosition().

В устройствах, включающих аппаратную поддержку GPS, имеется возможность определять местонахождение с высокой степенью точности с помощью устройства GPS. Однако чаще информация о местонахождении поступает из Всемирной паутины. Если броузер отправит IP-адрес специализированной веб-службе, она в большинстве случаев сможет определить (на основе информации о поставщиках услуг Интернета), в каком городе находится пользователь (и рекламодатели часто пользуются этой возможностью, реализуя определение местонахождения на стороне сервера). Броузер часто в состоянии получить еще более точную информацию о местонахождении, запросив у операционной системы список ближайших беспроводных сетей и силы их сигналов. Затем эта информация отправляется веб-службе, которая позволяет вычислить местонахождение с большой точностью (обычно с точностью до микрорайона в городе).

Эти технологии определения географического местонахождения связаны либо с обменом данными по сети, либо с взаимодействием с несколькими спутниками, поэтому прикладной интерфейс объекта Geolocation является асинхронным: методы getCurrentPosition() и watchPosition() возвращают управление немедленно, но они принимают функцию, которая будет вызвана броузером, когда он определит местонахождение пользователя (или когда местонахождение изменится). В простейшем случае запрос местонахождения выглядит так:

```
navigator.geolocation.getCurrentPosition(function(pos) {
    var latitude = pos.coords.latitude;
    var longitude = pos.coords.longitude;
    alert("Ваши координаты: " + latitude + ", " + longitude);
});
```

В дополнение к широте и долготе в ответ на каждый успешный запрос возвращается также значение (в метрах), указывающее точность определения местонахождения. Пример 22.1 демонстрирует получение информации о местонахождении: он вызывает метод getCurrentPosition(), чтобы определить текущее местонахождение, и использует полученную информацию для отображения карты (полученной от службы Google Maps) текущего местонахождения в масштабе, примерно соответствующем точности определения местонахождения.

Пример 22.1. Использование информации о местонахождении для отображения карты

```
// Возвращает вновь созданный элемент <img>, настроенный (в случае успешного
// определения местонахождения) на отображение карты для текущего местонахождения.
// Обратите внимание, что вызывающая программа сама должна вставить возвращаемый
// элемент в документ, чтобы отобразить его. Возбуждает исключение, если возможность
// определения местонахождения не поддерживается броузером.
function getmap() {
    // Проверить поддержку объекта geolocation
    if (!navigator.geolocation)
        throw "Определение местонахождения не поддерживается";

    // Создать новый элемент <img>, отправить запрос определения местонахождения,
    // чтобы в img отобразить карту местонахождения и вернуть изображение.
    var image = document.createElement("img");
    navigator.geolocation.getCurrentPosition(setMapURL);
    return image;

    // Эта функция будет вызвана после того, как вызывающая программа получит объект
    // изображения, в случае успешного выполнения запроса определения местонахождения.
    function setMapURL(pos) {
        // Получить информацию о местонахождении из объекта аргумента
        var latitude = pos.coords.latitude;   // Градусы к северу от экватора
        var longitude = pos.coords.longitude; // Градусы к востоку от Гринвича
        var accuracy = pos.coords.accuracy;   // Метры

        // Сконструировать URL для получения статического изображения карты
        // от службы Google Map для этого местонахождения
        var url = "http://maps.google.com/maps/api/staticmap" +
                  "?center=" + latitude + "," + longitude +
                  "&size=640x640&sensor=true";

        // Установить масштаб карты, используя грубое приближение
        var zoomlevel=20;  // Для начала установить самый крупный масштаб
        if (accuracy > 80) // Уменьшить масштаб для низкой точности
            zoomlevel -= Math.round(Math.log(accuracy/50)/Math.LN2);
        url += "&zoom=" + zoomlevel; // Добавить масштаб в URL

        // Отобразить карту в объекте изображения. Спасибо, Google!
        image.src = url;
    }
}
```

Прикладной интерфейс `Geolocation` обладает несколькими особенностями, которые не были продемонстрированы в примере 22.1:

- В дополнение к первому аргументу с функцией обратного вызова методы `getCurrentPosition()` и `watchPosition()` принимают вторую необязательную функцию, которая будет вызвана в случае неудачного выполнения запроса.
- Помимо функций обработчиков успешного и неудачного выполнения запроса эти два метода принимают в третьем необязательном аргументе объект с параметрами. Свойства этого объекта определяют: желательна ли высокая точность определения местонахождения, насколько «устаревшей» может быть информация о местонахождении и предельное время ожидания определения местонахождения.

- Объект, который передается обработчику в случае успешного выполнения запроса, также включает время и может (на некоторых устройствах) содержать дополнительную информацию, такую как высота над уровнем моря, скорость и направление перемещения.

Эти дополнительные возможности демонстрируются в примере 22.2.

Пример 22.2. Демонстрация всех возможностей определения местонахождения

```
// Асинхронно определяет местонахождение и отображает его в указанном элементе.
function whereami(elt) {
    // Этот объект передается методу getCurrentPosition() в 3 аргументе
    var options = {
        // Чтобы получить координаты с высокой точностью (например, с устройства GPS),
        // присвойте этому свойству значение true. Отметьте, однако, что это может
        // увеличить расход энергии в аккумуляторах.
        enableHighAccuracy: false, // Приблизительно: по умолчанию

        // Определите свое значение, если допустимо брать координаты из кэша.
        // По умолчанию имеет значение 0, что обеспечивает получение самой
        // свежей информации.
        maximumAge: 300000, // Пригодна информация, полученная в течение последних 5 минут

        // Предельное время ожидания выполнения запроса.
        // По умолчанию имеет значение Infinity, что соответствует бесконечному
        // времени ожидания выполнения запроса вызовом метода getCurrentPosition()
        timeout: 15000 // Ждать не более 15 секунд
    };

    if (navigator.geolocation) // Запросить координаты, если возможно
        navigator.geolocation.getCurrentPosition(success, error, options);
    else
        elt.innerHTML = "Возможность определения местонахождения " +
                        "не поддерживается этим броузером";

    // Эта функция будет вызвана в случае неудачного выполнения запроса
    function error(e) {
        // Объект ошибки содержит числовой код и текстовое сообщение. Коды:
        // 1: пользователь не дал разрешения на определение местонахождения
        // 2: броузер не смог определить местонахождение
        // 3: истекло предельное время ожидания
        elt.innerHTML = "Ошибка определения местонахождения " +
                        e.code + ": " + e.message;
    }

    // Эта функция будет вызвана в случае успешного выполнения запроса
    function success(pos) {
        // Эти поля возвращаются всегда. Обратите внимание, что поле timestamp
        // принадлежит внешнему объекту pos, а не вложенному coords.
        var msg = "At " +
            new Date(pos.timestamp).toLocaleString() + " вы находились в " +
            pos.coords.accuracy + " метрах от точки " +
            pos.coords.latitude + " северной широты " +
            pos.coords.longitude + " восточной долготы.";

        // Если устройство возвращает высоту над уровнем моря, добавить эту информацию.
        if (pos.coords.altitude) {
            msg += " Вы находитесь на высоте " + pos.coords.altitude + " ± " +
                pos.coords.altitudeAccuracy + " метров над уровнем моря.";
        }

        // Если устройство возвращает направление и скорость движения,
        // добавить и эту информацию.
        if (pos.coords.speed) {
            msg += " Вы перемещаетесь со скоростью " +
                pos.coords.speed + "м/сек в направлении " +
                pos.coords.heading + ".";
        }
        elt.innerHTML = msg; // Отобразить информацию о местонахождении
    }
}
```

22.2. Управление историей посещений

Веб-броузеры запоминают, какие документы загружались в окно, и предоставляют кнопки Back и Forward, позволяющие перемещаться между этими документами. Эта модель хранения истории посещений в броузерах появилась еще в те дни, когда документы были статическими и все вычисления выполнялись на стороне сервера. В настоящее время веб-приложения часто загружают содержимое динамически и отображают новые состояния приложения без полной перезагрузки документа. Такие приложения должны предусматривать собственные механизмы управления историей посещений, если необходимо дать пользователю возможность использовать кнопки Back и Forward для перехода из одного состояния приложения в другое интуитивно понятным способом. Спецификация HTML5 определяет два механизма управления историей посещений.

Простейший способ работы с историей посещений связан с использованием свойства `location.hash` и события «hashchange». На момент написания этих строк данный способ был также наиболее широко реализованным: его поддержка в броузерах появилась еще до того, как он был стандартизован спецификацией HTML5. В большинстве броузеров (кроме старых версий IE) изменение свойства `location.hash` приводит к изменению URL, отображаемого в строке ввода адреса, и добавлению записи в историю посещений. Свойство `hash` определяет идентификатор фрагмента в URL и традиционно использовалось для перемещения к разделу документа с указанным идентификатором. Но свойство `location.hash` не обязательно должно определять идентификатор элемента: в него можно записать произвольную строку. Если состояние приложения можно представить в виде строки, эту строку можно будет использовать как идентификатор фрагмента.

Предусмотрев изменение значения свойства `location.hash`, вы даете пользователю возможность использовать кнопки Back и Forward для перемещения между состояниями приложения. Чтобы такие перемещения были возможны, приложение должно иметь способ определять момент изменения состояния, прочитать строку, хранящуюся в виде идентификатора фрагмента, и обновить себя в соответствии с требуемым состоянием. Согласно спецификации HTML5, при изменении идентификатора фрагмента броузер должен возбуждать событие «hashchange» в объекте `Window`. В броузерах, поддерживающих событие «hashchange», можно присвоить свойству `window.onhashchange` функцию обработчика, которая будет вызываться при каждом изменении идентификатора фрагмента документа, вызванном перемещением по истории посещений. При вызове эта функция-обработчик должна проанализировать значение `location.hash` и отобразить содержимое страницы, соответствующее выбранному состоянию.

Спецификация HTML5 также определяет другой, более сложный и более надежный способ управления историей посещений, основанный на использовании метода `history.pushState()` и события «popstate». При переходе в новое состояние веб-приложение может вызвать метод `history.pushState()`, чтобы добавить это состояние в историю посещений. В первом аргументе методу передается объект, содержащий всю информацию, необходимую для восстановления текущего состояния приложения. Для этой цели подойдет любой объект, который можно преобразовать в строку вызовом метода `JSON.stringify()`, а также некоторые встроенные типы, такие как `Date` и `RegExp` (смотрите врезку ниже). Во втором аргументе передается необязательное заглавие (простая текстовая строка), которую броузер сможет использовать для идентификации сохраненного состояния в истории посещений (например, в меню кнопки Back). В третьем необязательном аргументе передается строка URL, которая будет отображаться как адрес текущего состояния. Относительные URL-адреса интерпретируются относительно текущего адреса документа и нередко определяют лишь часть URL, соответствующую идентификатору фрагмента, такую как `#state`. Связывание различных состояний приложения с собственными URL-адресами дает пользователю возможность делать закладки на внутренние состояния приложения, и если в строке URL будет указан достаточное количество информации, приложение сможет восстановить это состояние при запуске с помощью закладки.

Структурированные копии

Как отмечалось выше, метод `pushState()` принимает объект с информацией о состоянии и создает его частную копию. Это полная, глубокая копия объекта: при ее создании рекурсивно копируется содержимое всех вложенных объектов и массивов. В стандарте HTML5 такие копии называются *структурированными копиями*. Процедура создания структурированной копии напоминает передачу объекта функции `JSON.stringify()` и обработку результата функцией `JSON.parse()` (раздел 6.9). Но в формат JSON можно преобразовать только простые значения JavaScript, а также объекты и массивы. Стандарт HTML5 требует, чтобы алгоритм создания структурированных копий поддерживал возможность создания копий объектов `Date` и `RegExp`, `ImageData` (полученных из элементов `<canvas>` – раздел 21.4.14) и `FileList`, `File` и `Blob` (описывается в разделе 22.6). Функции JavaScript и объекты ошибок явно исключены из списка объектов, поддерживаемых алгоритмом создания структурированных копий, также как и большинство объектов среды выполнения, таких как окна, документы, элемент и т. д.

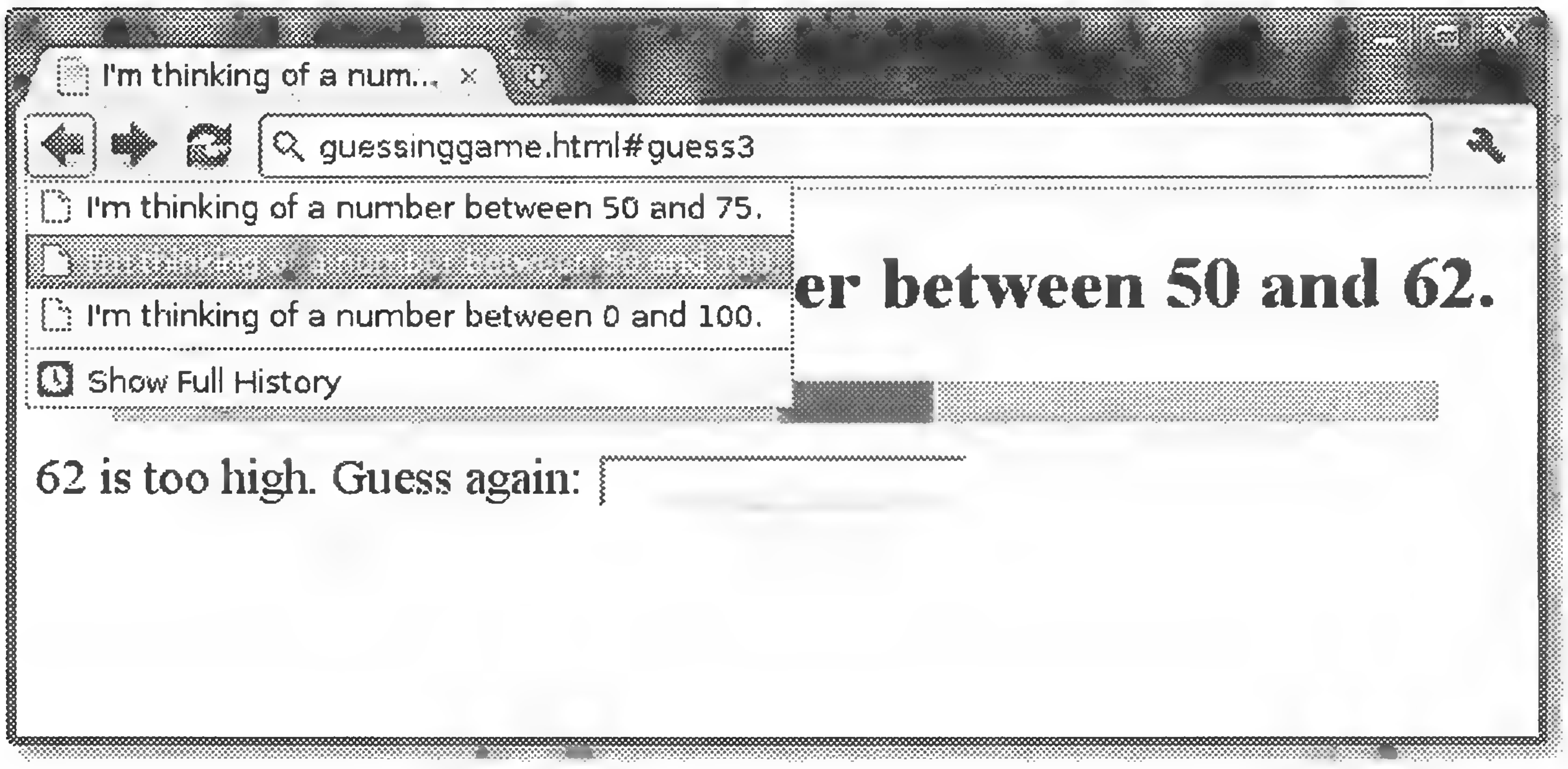
Вряд ли вам понадобится сохранять файлы или изображения как часть состояния приложения, однако структурированные копии также используются некоторыми другими стандартами, связанными со стандартом HTML5, и мы будем встречать их на протяжении всей этой главы.

В дополнение к методу pushState() объект History определяет метод replaceState(), который принимает те же аргументы, но не просто добавляет новую запись в историю посещений, а замещает текущую запись.

Когда пользователь перемещается по истории посещений с помощью кнопок Back и Forward, броузер возбуждает событие «popstate» в объекте Window. Объект, связанный с этим событием, имеет свойство с именем state, содержащее копию (еще одну структурированную копию) объекта с информацией о состоянии, переданного методу pushState().

В примере 22.3 демонстрируется простое веб-приложение – игра «Угадай число», изображенная на рис. 22.1, – в которой используются описанные приемы сохранения истории посещений, определяемые стандартом HTML5, с целью дать пользователю возможность «вернуться назад», чтобы пересмотреть или повторить попытку.

Когда эта книга готовилась к печати, в броузере Firefox 4 было внесено два изменения в прикладной интерфейс объекта History, которые могут быть заимствованы и другими броузерами. Во-первых, в Firefox 4 информация о текущем состоянии теперь доступна через свойство state самого объекта History, а это означает, что вновь загружаемые страницы не должны ждать события «popstate». Во-вторых, Firefox 4 более не возбуждает событие «popstate» для вновь загруженных страниц, для которых отсутствует сохраненное состояние. Это второе изменение означает, например, что пример, приведенный ниже, будет работать неправильно в Firefox 4.


I'm thinking of a num...
guessinggame.html#guess3
I'm thinking of a number between 50 and 75.
I'm thinking of a number between 0 and 100.
Show Full History

er between 50 and 62.

62 is too high. Guess again:

Рис. 22.1. Игра «Угадай число»

Примерх 22.3. Управление историей посещений с помощью pushState()

```
<!DOCTYPE html>
<html><head><title>I'm thinking of a number...</title>
<script>
window.onload = newgame;       // Начать новую игру при загрузке
window.onpopstate = popState; // Обработчик событий истории посещений
var state, ui;           // Глобальные переменные, инициализируемые в функции newgame()

function newgame(playagain) { // Начинает новую игру "Угадай число"
    // Настроить объект для хранения необходимых элементов документа
    ui = {
        heading: null, // Заголовок <h1> в начале документа.
        prompt: null,  // Текст предложения ввести число.
        input: null,   // Поле, куда пользователь вводит числа.
        low: null,     // Три ячейки таблицы для визуального представления
        mid: null,     // ...диапазона, в котором находится загаданное число.
        high: null
    };
    // Отыскать каждый из этих элементов по их атрибутам id
    for(var id in ui) ui[id] = document.getElementById(id);

    // Определить обработчик событий для поля ввода
    ui.input.onchange = handleGuess;

    // Выбрать случайное число и инициализировать состояние игры
    state = {
        n: Math.floor(99 * Math.random()) + 1, // Целое число: 0 < n < 100
        low: 0,          // Нижняя граница, где находится угадываемое число
        high: 100,       // Верхняя граница, где находится угадываемое число
        guessnum: 0,     // Количество выполненных попыток угадать число
        guess: undefined // Последнее число, указанное пользователем
    };

    // Изменить содержимое документа, чтобы отобразить начальное состояние
    display(state);

    // Эта функция вызывается как обработчик события onload, а также как обработчик щелчка
    // на кнопке Play Again (Играть еще раз), которая появляется в конце игры.
    // Во втором случае аргумент playagain будет иметь значение true, и если это так,
    // мы сохраняем новое состояние игры. Но если функция была вызвана в ответ
    // на событие "load", сохранять состояние не требуется, потому что событие "load"
    // может возникнуть также при переходе назад по истории посещений из какого-то
    // другого документа в существующее состояние игры. Если бы мы сохраняли начальное
    // состояние, в этом случае мы могли бы затереть имеющееся в истории актуальное
    // состояние игры. В броузерах, поддерживающих метод pushState(), за событием "load"
    // всегда следует событие "popstate". Поэтому, вместо того чтобы сохранять
    // состояние здесь, мы ждем событие "popstate". Если вместе с ним будет получен
    // объект состояния, мы просто используем его. Иначе, если событие "popstate"
    // содержит в поле state значение null, мы знаем, что была начата новая игра,
    // и поэтому используем replaceState для сохранения нового состояния игры.
    if (playagain === true) save(state);
}

// Сохраняет состояние игры с помощью метода pushState(), если поддерживается
function save(state) {
    if (!history.pushState) return;// Вернуться, если pushState() не определен

    // С каждым состоянием мы связываем определенную строку URL-адреса.
    // Этот адрес отображает число попыток, но не содержит информации о состоянии игры,
    // поэтому его нельзя использовать для создания закладок.
    // Мы не можем поместить информацию о состоянии в URL-адрес,
    // потому что при этом пришлось бы указать в нем угадываемое число.
    var url = "#guess" + state.guessnum;
    // Сохранить объект с информацией о состоянии и строку URL
    history.pushState(state, // Сохраняемый объект с информацией о состоянии
                      "",    // Заглавие: текущие броузеры игнорируют его
                      url);  // URL состояния: бесполезен для закладок
}
```

```
// Обработчик события onpopstate, восстанавливающий состояние приложения.
function popState(event) {
    if (event.state) { // Если имеется объект состояния, восстановить его
        // Обратите внимание, что event.state является полной копией
        // сохраненного объекта состояния, поэтому мы можем изменять его,
        // не опасаясь изменить сохраненное значение.
        state = event.state; // Восстановить состояние
        display(state);      // Отобразить восстановленное состояние
    }
    else {
        // Когда страница загружается впервые, событие "popstate" поступает
        // без объекта состояния. Заменить значение null действительным
        // состоянием: смотрите комментарий в функции newgame().
        // Нет необходимости вызывать display() здесь.
        history.replaceState(state, "", "#guess" + state.guessnum);
    }
};

// Этот обработчик событий вызывается всякий раз, когда пользователь вводит число.
// Он обновляет состояние игры, сохраняет и отображает его.
function handleGuess() {
    // Извлечь число из поля ввода
    var g = parseInt(this.value);
    // Если это число и оно попадает в требуемый диапазон
    if ((g > state.low) && (g < state.high)) {
        // Обновить объект состояния для этой попытки
        if (g < state.n) state.low = g;
        else if (g > state.n) state.high = g;
        state.guess = g;
        state.guessnum++;
        // Сохранить новое состояние в истории посещений
        save(state);
        // Изменить документ в ответ на попытку пользователя
        display(state);
    }
    else { // Ошибочная попытка: не сохранять новое состояние
        alert("Please enter a number greater than " + state.low +
              " and less than " + state.high);
    }
}

// Изменяет документ, отображая текущее состояние игры.
function display(state) {
    // Отобразить заголовок документа
    ui.heading.innerHTML = document.title =
        "I'm thinking of a number between " +
        state.low + " and " + state.high + ".";

    // Отобразить диапазон чисел с помощью таблицы
    ui.low.style.width = state.low + "%";
    ui.mid.style.width = (state.high-state.low) + "%";
    ui.high.style.width = (100-state.high) + "%";

    // Сделать поле ввода видимым, очистить его и установить фокус ввода
    ui.input.style.visibility = "visible";
    ui.input.value = "";
    ui.input.focus();
```

```
        // Вывести приглашение к вводу, опираясь на последнюю попытку
        if (state.guess === undefined)
            ui.prompt.innerHTML = "Type your guess and hit Enter: ";
        else if (state.guess < state.n)
            ui.prompt.innerHTML = state.guess + " is too low. Guess again: ";
        else if (state.guess > state.n)
            ui.prompt.innerHTML = state.guess + " is too high. Guess again: ";
        else {
            // Если число угадано, скрыть поле ввода и отобразить кнопку
            // Play Again (Играть еще раз).
            ui.input.style.visibility = "hidden"; // Попыток больше не будет
            ui.heading.innerHTML = document.title = state.guess + " is correct! ";
            ui.prompt.innerHTML =
                "You Win! <button onclick='newgame(true)'>Play Again</button>";
        }
    }
</script>
<style> /* CSS-стили, чтобы придать игре привлекательный внешний вид */
#prompt { font-size: 16pt; }
table { width: 90%; margin:10px; margin-left:5%; }
#low, #high { background-color: lightgray; height: 1em; }
#mid { background-color: green; }
</style>
</head>
<body><!-- Следующие элементы образуют пользовательский интерфейс игры -->
<!-- Заголовок игры и текстовое представление диапазона чисел -->
<h1 id="heading">I'm thinking of a number...</h1>
<!-- визуальное представление чисел, которые еще не были исключены -->
<table><tr><td id="low"></td><td id="mid"></td><td id="high"></td></tr></table>
<!-- Поле ввода чисел -->
<label id="prompt"></label><input id="input" type="text">
</body></html>
```

22.3. Взаимодействие документов с разным происхождением

Как отмечалось в разделе 14.8, некоторые окна и вкладки броузера полностью изолированы друг от друга, и сценарий, выполняющийся в одном окне или вкладке, ничего не знает о существовании других окон и вкладок. В других случаях, когда сценарий сам открывает новые окна или работает с вложенными фреймами, сценарии в этих окнах и фреймах могут быть осведомлены друг о друге. Если эти окна или фреймы содержат документы, полученные от одного и того же веб-сервера, сценарии в этих окнах и фреймах смогут взаимодействовать друг с другом и манипулировать документами друг друга.

Однако иногда сценарий имеет возможность сослаться на другой объект `Window`, но, поскольку содержимое этого окна имеет иное происхождение, веб-броузер (следующий требованиям политики общего происхождения) не позволит сценарию просматривать содержимое документа в этом другом окне. Броузер не позволит сценарию читать значения свойств или вызывать методы другого окна. Единственный метод окна, который *может* вызвать сценарий из документа с другим происхождением, называется `postMessage()`, и этот метод обеспечивает возможность ограниченных взаимодействий – в форме асинхронных сообщений – между сценариями из документов с разным происхождением. Этот вид взаимодействий определяется стандартом HTML5 и реализован во всех текущих броузерах (включая IE8 и более поздние версии). Этот прием известен как «обмен сообщениями между документами с разным происхождением», но, так как прикладной интерфейс поддержки определен в объекте `Window`, а не в объекте документа, возможно, лучше было бы назвать этот прием «обменом сообщениями между окнами».

Метод `postMessage()` принимает два обязательных аргумента. Первый – передаваемое сообщение. Согласно спецификации HTML5, это может быть любое простое значение или объект, который можно скопировать (смотрите врезку «Структурированные копии» выше), но некоторые текущие броузеры (включая Firefox 4 beta) принимают только строки, поэтому, если в сообщении потребуется передать объект или массив, его необходимо будет предварительно сериализовать с помощью функции `JSON.stringify()` (раздел 6.9).

Во втором аргументе должна передаваться строка, определяющая ожидаемое происхождение окна, принимающего сообщение. Эта строка должна включать протокол, имя хоста и (необязательно) номер порта (можно передать полный URL-адрес, но все, кроме протокола, хоста и порта будет игнорироваться). Это предусмотрено из соображений безопасности: злонамеренный программный код или обычные пользователи могут открывать новые окна с документами, взаимодействие с которыми вы не предполагали, поэтому `postMessage()` не будет доставлять ваши сообщения, если окно будет содержать документ с происхождением, отличным от указанного. Если передаваемое сообщение не содержит конфиденциальной информации и вы готовы передать ее сценарию из документа с любым происхождением, то во втором аргументе можно передать строку `"*"`, которая будет играть роль шаблонного символа. Если необходимо указать, что документ должен иметь то же происхождение, что и документ в текущем окне, можно передать строку `"/"`.

Если происхождение документа совпадет с указанным в аргументе, вызов метода `postMessage()` приведет к возбуждению события «message» в целевом объекте `Window`. Сценарий в этом окне может определить обработчик для обработки событий «message», которому будет передан объект события со следующими свойствами:

`data`

Копия сообщения, переданного методу `postMessage()` в первом аргументе.

`source`

Объект `Window`, из которого было отправлено сообщение.

`origin`

Строка, определяющая происхождение (в виде URL) документа, отправившего сообщение.

Большинство обработчиков `onmessage()` должно сначала проверить свойство `origin` своего аргумента и должно игнорировать сообщения из неподдерживаемых доменов.

Обмен сообщениями с документами из сторонних доменов посредством метода `postMessage()` и обработки события «message» может пригодиться, например, когда необходимо подключить к веб-странице модуль, или «гаджет», с другого сайта. Если модуль достаточно прост и автономен, его можно просто загрузить в элемент `<iframe>`. Однако представьте, что это более сложный модуль, который опре-

деляет свой прикладной интерфейс, и ваша веб-страница должна управлять этим модулем или как-то иначе взаимодействовать с ним. Если модуль определен в виде элемента `<script>`, он сможет предложить полноценный прикладной интерфейс на языке JavaScript, но, с другой стороны, включение его в страницу даст ему полный контроль над страницей и ее содержимым. Такой подход широко используется в современной Всемирной паутине (особенно в веб-рекламе), но в действительности это не самое лучшее решение, даже если вы доверяете другому сайту.

Альтернативой является обмен сообщениями между документами с разным происхождением: автор модуля может упаковать его в HTML-файл, который принимает события «message» и передает их для обработки соответствующим функциям на языке JavaScript. В этом случае веб-страница, подключившая модуль, сможет взаимодействовать с ним, отправляя сообщения с помощью метода `postMessage()`. Этот прием демонстрируется в примерах 22.4 и 22.5. В примере 22.4 приводится реализация простого модуля, подключаемого с помощью элемента `<iframe>`, который выполняет поиск на сайте Twitter и отображает сообщения, соответствующие указанной фразе. Чтобы выполнить поиск с помощью этого модуля, подключившая его страница должна просто отправить ему сообщение с требуемой фразой.

Пример 22.4. Модуль поиска на сайте Twitter с помощью метода postMessage()

```
<!DOCTYPE html>
<!--
  Это модуль поиска на сайте Twitter. Модуль можно подключить к любой странице внутри
  элемента <iframe> и запросить его выполнить поиск, отправив ему строку запроса
  с помощью метода postMessage(). Поскольку модуль подключается в элементе <iframe>,
  а не <script>, он не сможет получить доступ к содержимому вмещающего документа.
-->
<html>
<head>
<style>body { font: 9pt sans-serif; }</style>
<!-- Подключить библиотеку jQuery ради ее утилиты jQuery.getJSON() -->
<script src="http://code.jquery.com/jquery-1.4.4.min.js"/></script>
<script>
// Можно было бы просто использовать свойство window.onmessage,
// но некоторые старые броузеры (такие как Firefox 3) не поддерживают его,
// поэтому обработчик устанавливается таким способом.
if (window.addEventListener)
    window.addEventListener("message", handleMessage, false);
else
    window.attachEvent("onmessage", handleMessage); // Для IE8

function handleMessage(e) {
    // Нас не волнует происхождение документа, отправившего сообщение:
    // мы готовы выполнить поиск на сайте Twitter для любой страницы.
    // Однако сообщение должно поступить от окна, вмещающего этот модуль.
    if (e.source !== window.parent) return;

    var searchterm = e.data; // Это фраза, которую требуется отыскать

    // С помощью утилит поддержки Ajax из библиотеки jQuery и прикладного
    // интерфейса Twitter отыскать сообщения, соответствующие фразе.
    jQuery.getJSON("http://search.twitter.com/search.json?callback=?",
                   { q: searchterm },
                   function(data) { // Вызывается с результатами запроса
```

```
                        var tweets = data.results;
                        // Создать HTML-документ для отображения результатов
                        var escaped = searchterm.replace("<", "&lt;");
                        var html = "<h2>" + escaped + "</h2>";
                        if (tweets.length == 0) {
                            html += "No tweets found";
                        }
                        else {
                            html += "<dl>"; // <dl> list of results
                            for(var i = 0; i < tweets.length; i++) {
                                var tweet = tweets[i];
                                var text = tweet.text;
                                var from = tweet.from_user;
                                var tweeturl = "http://twitter.com/#!/" +
                                    from + "/status/" + tweet.id_str;
                                html += "<dt><a target='_blank' href='" +
                                    tweeturl + "'>" + tweet.from_user +
                                    "</a></dt><dd>" + tweet.text + "</dd>";
                            }
                            html += "</dl>";
                        }
                        // Вставить документ в <iframe>
                        document.body.innerHTML = html;
                    });
}
$(function() {
    // Сообщить вмещающему документу, что модуль готов к поиску. Вмещающий документ
    // не может отправлять модулю сообщения до получения этого сообщения, потому что
    // модуль еще не готов принимать сообщения. Вмещающий документ может просто
    // дождаться события onload, чтобы определить момент, кода будут загружены
    // все фреймы. Но мы предусматриваем отправку этого сообщения, чтобы известить
    // вмещающий документ, что можно начать выполнять поиск еще до получения
    // события onload. Модулю неизвестно происхождение вмещающего документа,
    // поэтому мы используем *, чтобы броузер доставил сообщение любому документу.
    window.parent.postMessage("Twitter Search v0.1", "*");
});
</script>
</head>
<body>
</body>
</html>
```

В примере 22.5 представлен простой сценарий на языке JavaScript, который может быть вставлен в любую веб-страницу, где требуется использовать модуль поиска по сайту Twitter. Он вставляет модуль в документ и затем устанавливает обработчики событий во все ссылки в документе, чтобы при наведении указателя мыши на ссылку производился вызов метода `postMessage()` **модуля, заставляющий его выполнить поиск по строке URL ссылки. Это позволит узнать, что говорят люди о веб-сайте перед тем, как перейти к нему.**

Пример 22.5. Использование модуля поиска по сайту Twitter с помощью метода postMessage()

```
// Этот сценарий JS вставляет Twitter Search Gadget в документ и добавляет обработчик
// событий ко всем ссылкам в документе, чтобы при наведении указателя мыши на них
// с помощью модуля поиска отыскать упоминания URL-адресов в ссылках.
// Это позволяет узнать, что говорят другие о сайтах, прежде чем щелкнуть на ссылке.
window.addEventListener("load", function() { // Не работает в IE < 9
    var origin = "http://davidflanagan.com";       // Происхождение модуля
    var gadget = "/demos/TwitterSearch.html";      // Путь к модулю
    var iframe = document.createElement("iframe"); // Создать iframe
    iframe.src = origin + gadget;                  // Установить его URL
    iframe.width = "250";                          // Ширина 250 пикселов
    iframe.height = "100%";                        // Во всю высоту документа
    iframe.style.cssFloat = "right";               // Разместить справа

    // Вставить фрейм в начало документа
    document.body.insertBefore(iframe, document.body.firstChild);

    // Отыскать все ссылки и связать их с модулем
    var links = document.getElementsByTagName("a");
    for(var i = 0; i < links.length; i++) {
        // addEventListener не работает в версии IE8 и ниже
        links[i].addEventListener("mouseover", function() {
            // Отправить url как искомую фразу и доставлять его, только если
            // фрейм все еще отображает документ с сайта davidflanagan.com
            iframe.contentWindow.postMessage(this.href, origin);
        }, false);
    }
}, false);
```

22.4. Фоновые потоки выполнения

Одна из основных особенностей клиентских сценариев на языке JavaScript заключается в том, что они выполняются в единственном потоке выполнения: броузер, к примеру, никогда не будет выполнять два обработчика событий одновременно, и таймер никогда не сработает, пока выполняется обработчик события. Параллельные обновления переменных приложения или документа просто невозможны, и программистам, разрабатывающим клиентские сценарии, не требуется задумываться или хотя бы понимать особенности параллельного программирования. Как следствие, функции в клиентских сценариях на языке JavaScript должны выполняться очень быстро, иначе они остановят работу цикла событий и веб-броузер перестанет откликаться на действия пользователя. Именно по этой причине прикладной интерфейс поддержки Ajax всегда действует асинхронно, и по этой же причине в клиентском JavaScript отсутствуют синхронные версии функций `load()` или `require()` для загрузки библиотек на языке JavaScript.

Спецификация «Web Workers»[1] со всеми мерами предосторожности ослабляет ограничение на единственный поток выполнения в клиентском JavaScript. «Фоновые потоки», определяемые спецификацией, фактически являются параллельными потоками выполнения. Однако эти потоки выполняются в изолированной среде, не имеют доступа к объектам `Window` и `Document` и могут взаимодействовать с основным потоком выполнения только посредством передачи асинхронных сообщений. Это означает, что параллельные изменения дерева DOM по-прежнему невозможны, но это также означает, что теперь имеется возможность использовать синхронные прикладные интерфейсы и писать функции, выполняющиеся длительное время, которые не будут останавливать работу цикла событий и подвешивать броузер. Создание нового фонового потока выполнения не является такой тяжеловесной операцией, как открытие нового окна броузера, но также не является легковесной операцией, вследствие чего нет смысла создавать новые фоновые потоки для выполнения тривиальных операций. В сложных веб-приложениях может оказаться полезным применение даже нескольких десятков фоновых потоков, но весьма маловероятно, что приложения с сотнями и тысячами таких потоков смогут иметь практическую ценность.

---

[1] Спецификация «Web Workers» изначально была частью спецификации HTML5, но затем была выделена в отдельный, независимый, хотя и тесно связанный со стандартом, документ. На момент написания этих строк проект спецификации был доступен по адресам *http://dev.w3.org/html5/workers/* и *http://whatwg.org/ww*.

Как и любой прикладной интерфейс поддержки многопоточных приложений, спецификация «Web Workers» определяет две различные части. Первая – объект `Worker`, который представляет фоновый поток выполнения в программе, создавшей его. Вторая – объект `WorkerGlobalScope`, глобальный объект нового фонового потока выполнения, который представляет фоновый поток выполнения внутри него самого. Оба объекта описываются в следующих подразделах. За ними следует раздел с примерами.

22.4.1. Объект Worker

Чтобы создать новый фоновый поток, достаточно просто вызвать конструктор `Worker()`, передав ему URL-адрес, определяющий программный код на языке JavaScript, который должен выполняться в фоновом потоке:

```
var loader = new Worker("utils/loader.js");
```

Если указать относительный URL-адрес, он будет интерпретироваться относительно URL-адреса документа, содержащего сценарий, который вызвал конструктор `Worker()`. Если указать абсолютный URL-адрес, он должен иметь то же происхождение (протокол, имя хоста и порт), что и вмещающий документ.

После создания объекта `Worker` ему можно отправлять данные с помощью его метода `postMessage()`. Значение, переданное методу `postMessage()`, будет скопировано (смотрите врезку «Структурированные копии» выше), и полученная копия будет передана фоновому потоку вместе с событием «message»:

```
loader.postMessage("file.txt");
```

Обратите внимание, что, в отличие от метода `postMessage()` объекта `Window`, метод `postMessage()` объекта `Worker` не имеет аргумента, в котором передавалась бы строка, описывающая происхождение (раздел 22.3). Кроме того, метод `postMessage()` объекта `Worker` корректно копирует сообщение во всех текущих броузерах, в отличие от `Window.postMessage()`, который в некоторых основных броузерах способен принимать только строковые сообщения.

Принимать сообщения от фонового потока можно с помощью обработчика события «message» объекта `Worker`:

```
worker.onmessage = function(e) {
    var message = e.data;                     // Извлечь сообщение
    console.log("Содержимое: " + message);    // Выполнить некоторые действия
}
```

Если фоновый поток возбудит исключение и не обработает его, это исключение продолжит распространение в виде события, которое также можно перехватить:

```
worker.onerror = function(e) {
    // Вывести текст ошибки, включая имя файла фонового потока и номер строки
    console.log("Ошибка в " + e.filename + ":" + e.lineno + ": " + e.message);
}
```

Подобно всем объектам, в которых могут возбуждаться события, объект `Worker` определяет стандартные методы `addEventListener()` и `removeEventListener()`, которые можно использовать вместо свойств `onmessage` и `onerror`, если необходимо установить несколько обработчиков событий.

Объект `Worker` имеет еще один метод, `terminate()`, который останавливает выполнение фонового потока.

22.4.2. Область видимости фонового потока

При создании нового фонового потока с помощью конструктора `Worker()` вы задаете URL-адрес файла с программным кодом на языке JavaScript. Этот программный код выполняется в новой, нетронутой среде выполнения JavaScript, полностью изолированной от сценария, запустившего фоновый поток. Глобальным объектом этой изолированной среды выполнения является объект `WorkerGlobalScope`. Объект `WorkerGlobalScope` – это чуть больше, чем просто глобальный объект JavaScript, но меньше, чем полноценный клиентский объект `Window`.

Объект `WorkerGlobalScope` имеет метод `postMessage()` и свойство обработчика события `onmessage`, подобные своим аналогам в объекте `Worker`, но действующие в обратном направлении. Вызов метода `postMessage()` внутри фонового потока сгенерирует событие «message» за его пределами, а сообщения, отправляемые извне, будут превращаться в события и передаваться обработчику `onmessage`. Обратите внимание: благодаря тому, что объект `WorkerGlobalScope` является глобальным объектом для фонового потока, метод `postMessage()` и свойство `onmessage` воспринимаются программным кодом, выполняющимся в фоновом потоке, как глобальная функция и глобальная переменная.

Функция `close()` позволяет фоновому потоку завершить свою работу, и по своему действию она напоминает метод `terminate()` объекта `Worker`. Отметьте, однако, что в объекте `Worker` нет метода, который позволил бы определить, прекратил ли работу фоновый поток, как и нет свойства обработчика события `onclose`. Если попытаться с помощью метода `postMessage()` передать сообщение фоновому потоку, который прекратил работу, это сообщение будет просто проигнорировано без возбуждения исключения. В общем случае, если фоновый поток может завершить работу самостоятельно вызовом метода `close()`, неплохо было бы предусмотреть отправку сообщения, извещающего о прекращении работы.

Наибольший интерес представляет глобальная функция `importScripts()`, определяемая объектом `WorkerGlobalScope`: фоновые потоки выполнения могут использовать ее для загрузки любых необходимых им библиотек. Например:

```
// Перед началом работы загрузить необходимые классы и утилиты
importScripts("collections/Set.js", "collections/Map.js", "utils/base64.js");
```

Функция `importScripts()` принимает один или более аргументов с URL-адресами, каждый из которых должен ссылаться на файл с программным кодом на языке

JavaScript. Относительные URL-адреса интерпретируются относительно URL-адреса, переданного конструктору `Worker()`. Она загружает и выполняет указанные файлы один за другим в том порядке, в каком они были указаны. Если при загрузке или при выполнении сценария возникнет какая-либо ошибка, ни один из последующих сценариев не будет загружаться или выполняться. Сценарий, загруженный функцией `importScripts()`, сам может вызвать функцию `importScripts()`, чтобы загрузить необходимые ему файлы. Отметьте, однако, что функция `importScripts()` не запоминает, какие сценарии были загружены и не предусматривает защиту от циклических ссылок.

Функция `importScripts()` выполняется синхронно: она не вернет управление, пока не будут загружены и выполнены все сценарии. Сценарии, указанные в вызове функции `importScripts()`, можно использовать сразу, как только она вернет управление: нет никакой необходимости определять функцию обратного вызова или обработчик события. После того как вы свыклись с асинхронной природой клиентского JavaScript, такой возврат к простой, синхронной модели может показаться странным. Но в этом и заключается достоинство потоков выполнения: в фоновом потоке можно использовать блокирующие функции, не блокируя цикл событий в основном потоке выполнения и не блокируя вычисления, выполняемые параллельно в других фоновых потоках.

Поскольку для фоновых потоков выполнения `WorkerGlobalScope` является глобальным объектом, он обладает всеми свойствами базового глобального объекта JavaScript, такими как объект `JSON`, функция `isNaN()` и конструктор `Date()`. (Полный список можно найти в справочной статье `Global`, в третьей части книги.) Однако, кроме того, объект `WorkerGlobalScope` имеет следующие свойства клиентского объекта `Window`:

- `self` – ссылка на сам глобальный объект. Отметьте, однако, что в объекте `WorkerGlobalScope` отсутствует синонимичное свойство `window`, имеющееся в объекте `Window`.

Модель выполнения фонового потока

Фоновые потоки выполняют свой программный код (и все импортированные сценарии) синхронно, от начала до конца, и затем переходят в асинхронную фазу выполнения, когда они откликаются на события и таймеры. Если фоновый поток регистрирует обработчик события `onmessage`, он никогда не завершит работу, пока есть вероятность поступления событий «message». Но если фоновый поток не принимает сообщения, он будет работать, пока не будут выполнены все задания (такие как загрузка или таймеры) и не будут вызваны все функции, связанные с этими заданиями. После вызова всех зарегистрированных функций обратного вызова в фоновом потоке нет никакой возможности начать выполнять новые задания, поэтому он может смело завершить свою работу. Представьте себе фоновый поток без обработчика событий `onmessage`, который загружает файл с помощью объекта `XMLHttpRequest`. Если обработчик `onload` запустит загрузку другого файла или зарегистрирует обработчик таймера вызовом функции `setTimeout()`, поток выполнения получит новое задание и продолжит работу. Иначе он завершится.

- Методы работы с таймерами `setTimeout()`, `clearTimeout()`, `setInterval()` и `clearInterval()`.
- Свойство `location`, содержащее URL-адрес, переданный конструктору `Worker()`. Это свойство ссылается на объект `Location`, как и аналогичное ему свойство `location` объекта `Window`. Объект `Location` имеет свойства `href`, `protocol`, `host`, `hostname`, `port`, `pathname`, `search` и `hash`. В фоновом потоке эти свойства доступны только для чтения.
- Свойство `navigator`, ссылающееся на объект, обладающий теми же свойствами, что и объект `Navigator` окна. Объект `navigator` фонового потока имеет свойства `appName`, `appVersion`, `platform`, `userAgent` и `onLine`.
- Обычные методы объектов, в которых могут возбуждаться события: `addEventListener()` и `removeEventListener()`.
- Свойство `onerror`, которому можно присвоить функцию обработки ошибок, подобное свойству `Window.onerror`, которое описывается в разделе 14.6. Обработчик ошибок, если его зарегистрировать, будет получать три аргумента с сообщением об ошибке, URL-адресом и номером строки. Он может вернуть `false`, чтобы показать, что ошибка обработана и не должна распространяться дальше в виде события «error» в объекте `Worker`. (Однако на момент написания этих строк обработка ошибок в разных броузерах была реализована несовместимым способом.)

Дополнительные особенности объекта Worker

Фоновые потоки выполнения, описываемые в этом разделе, являются *выделенными фоновыми потоками*: они связаны (или выделены из) с общим родительским потоком выполнения. Спецификация «Web Workers» определяет еще один тип фоновых потоков выполнения, *разделяемые потоки выполнения*. На момент написания этих строк броузеры еще не поддерживали разделяемые потоки выполнения. Однако их назначение состоит в том, чтобы играть роль именованного ресурса, который может предоставлять вычислительные услуги любым другим потокам выполнения. На практике взаимодействие с разделяемым потоком выполнения напоминает взаимодействие с сервером посредством сетевых сокетов.

Роль «сокета» для разделяемого потока выполнения играет объект `MessagePort`. Он определяет прикладной интерфейс обмена сообщениями, подобный аналогичному интерфейсу в выделенных потоках выполнения, а также используемому для обмена сообщениями между документами с разным происхождением. Объект `MessagePort` имеет метод `postMessage()` и атрибут `onmessage` обработчика событий. Спецификация HTML5 предусматривает возможность создания связанных пар объектов `MessagePort` с помощью конструктора `MessageChannel()`. Вы можете передавать объекты `MessagePort` (в специальном аргументе метода `postMessage()`) другим окнам или другим фоновым потокам выполнения и использовать их как выделенные каналы связи. Объекты `MessagePort` и `MessageChannel` являются дополнительным прикладным интерфейсом, который поддерживается лишь немногими броузерами и здесь не рассматривается.

Наконец, объект `WorkerGlobalScope` включает конструкторы объектов, важных для клиентских сценариев. В их числе конструктор `XMLHttpRequest()`, позволяющий фоновым потокам выполнять HTTP-запросы (глава 18), и конструктор `Worker()`, дающий возможность фоновым потокам создавать свои фоновые потоки. (Однако на момент написания этих строк конструктор `Worker()` был недоступен фоновым потокам в броузерах Chrome и Safari.)

Некоторые прикладные интерфейсы HTML5, описываемые далее в этой главе, определяют особенности, доступные как через обычный объект `Window`, так и через объект `WorkerGlobalScope`. Часто асинхронному прикладному интерфейсу объекта `Window` соответствует его синхронная версия в объекте `WorkerGlobalScope`. Эти прикладные интерфейсы «с поддержкой фоновых потоков выполнения» мы рассмотрим далее в этой главе.

22.4.3. Примеры использования фоновых потоков

Завершают этот раздел два примера использования фоновых потоков выполнения. Первый демонстрирует, как реализовать выполнение длительных вычислений в фоновом потоке, чтобы они не влияли на отзывчивость пользовательского интерфейса, обслуживаемого основным потоком выполнения. Второй демонстрирует, как можно использовать фоновые потоки для работы с простейшими синхронными прикладными интерфейсами.

В примере 22.6 определяется функция `smear()`, которая принимает элемент `<img>` в виде аргумента. Она применяет эффект размытия для «смазывания» изображения вправо. Для реализации этого эффекта используется описанный в главе 21 прием копирования изображения в неотображаемый элемент `<canvas>` и последующего извлечения пикселей изображения с помощью объекта `ImageData`. Элементы `<img>` и `<canvas>` нельзя передать фоновому потоку выполнения с помощью метода `postMessage()`, но можно передать объект `ImageData` (подробности во врезке «Структурированные копии» выше). Пример 22.6 создает объект `Worker` и вызывает его метод `postMessage()`, чтобы передать ему изображение. Когда фоновый поток отправит обработанные пикселы изображения обратно, программный код скопирует их снова в элемент `<canvas>`, извлекая их как ресурс с URL-адресом вида `data://` и устанавливая этот URL-адрес в качестве значения свойства `src` оригинального элемента `<img>`.

Пример 22.6. Создание фонового потока выполнения для обработки изображения

```
// Асинхронная замена изображения его смазанной версией.
// Используется так: <img src="testimage.jpg" onclick="smear(this)"/>
function smear(img) {
    // Создать неотображаемый элемент <canvas> того же размера, что и изображение
    var canvas = document.createElement("canvas");
    canvas.width = img.width;
    canvas.height = img.height;

    // Скопировать изображение в холст и извлечь его пикселы
    var context = canvas.getContext("2d");
    context.drawImage(img, 0, 0);
    var pixels = context.getImageData(0,0,img.width,img.height)

    // Отправить пикселы фоновому потоку выполнения
    var worker = new Worker("SmearWorker.js"); // Создать фоновый поток
```

```
        worker.postMessage(pixels);              // Скопировать и отдать пикселы

        // Зарегистрировать обработчик для получения ответа от фонового потока
        worker.onmessage = function(e) {
            var smeared_pixels = e.data;        // Пикселы, полученные от потока
            context.putImageData(smeared_pixels, 0, 0); // Скопировать в холст
            img.src = canvas.toDataURL();             // А затем в изображение
            worker.terminate();                       // Остановить поток
            canvas.width = canvas.height = 0;         // Освободить память
        }
    }
```

В примере 22.7 приводится программный код реализации фонового потока, используемого в примере 22.6. Основу этого примера составляет функция обработки изображения: модифицированная версия примера 21.10. Обратите внимание, что этот пример настраивает свою инфраструктуру обмена сообщениями единственной строчкой программного кода: обработчик события `onmessage` **просто накладывает эффект смазывания на изображение и сразу же отправляет его обратно.**

Пример 22.7. Обработка изображения в фоновом потоке выполнения

```
// Получает объект ImageData от основного потока выполнения, обрабатывает его
// и отправляет обратно
onmessage = function(e) { postMessage(smear(e.data)); }

// Смазывает пикселы в ImageData вправо, воспроизводя эффект быстрого движения.
// При обработке больших изображений этой функции приходится выполнять огромный объем
// вычислений, что может вызвать эффект подвисания пользовательского интерфейса,
// если использовать ее в основном потоке выполнения.
function smear(pixels) {
    var data = pixels.data, width = pixels.width, height = pixels.height;
    var n = 10, m = n-1; // Чем больше n, тем сильнее эффект смазывания
    for(var row = 0; row < height; row++) {          // Для каждой строки
        var i = row*width*4 + 4;                     // Индекс 2-го пиксела
        for(var col = 1; col < width; col++, i += 4) { // Для каждого столбца
            data[i] = (data[i] + data[i-4]*m)/n;     // Красная составляющая
            data[i+1] = (data[i+1] + data[i-3]*m)/n; // Зеленая
            data[i+2] = (data[i+2] + data[i-2]*m)/n; // Синяя
            data[i+3] = (data[i+3] + data[i-1]*m)/n; // Альфа-составляющая
        }
    }
    return pixels;
}
```

Обратите внимание, что программный код в примере 22.7 может обрабатывать любое количество изображений, которые будут отправлены ему. Однако для простоты пример 22.6 создает новый объект `Worker` **для обработки каждого изображения. Чтобы не плодить фоновые потоки, которые ничего не делают в ожидании новых сообщений, по завершении обработки изображения работа фонового потока завершается вызовом метода** `terminate()`.

Следующий пример демонстрирует, как с помощью фоновых потоков выполнения можно писать синхронный программный код и безопасно использовать его в клиентских сценариях на языке JavaScript. В разделе 18.1.2.1 было показано, как с помощью объекта `XMLHttpRequest` **выполнять синхронные HTTP-запросы,**

и говорилось, что такой способ его использования в основном потоке выполнения является не лучшим решением. Однако в фоновом потоке вполне оправданно выполнять синхронные запросы, и в примере 22.8 демонстрируется реализация фонового потока выполнения, которая выполняет именно такие запросы. Его обработчик события `onmessage` принимает массив URL-адресов, использует синхронный прикладной интерфейс объекта `XMLHttpRequest` для извлечения их содержимого и затем посылает полученное текстовое содержимое в виде массива строк обратно основному потоку выполнения. Или, если какой-либо HTTP-запрос потерпит неудачу, возбуждает исключение, которое распространится до обработчика `onerror` объекта `Worker`.

Отладка фоновых потоков выполнения

Одним из прикладных интерфейсов, недоступных в объекте `WorkerGlobalScope` (по крайней мере, на момент написания этих строк), является прикладной интерфейс доступа к консоли и одна из самых ценных его функций – `console.log()`. Фоновые потоки не могут выводить текст в консоль и вообще не могут взаимодействовать с документом, поэтому их отладка может оказаться весьма трудным делом. Если фоновый поток возбудит исключение, основной поток получит событие «error» в объекте `Worker`. Но чаще бывает необходимо иметь в фоновом потоке хоть какой-нибудь способ выводить отладочные сообщения, которые будут видимы в веб-консоли броузера. Один из самых простых способов добиться этого – изменить протокол передачи сообщений, используемый для взаимодействия с фоновым потоком, чтобы он мог посылать отладочные сообщения. Так, в примере 22.6 можно было бы вставить следующий программный код в начало обработчика событий `onmessage`:

```
if (typeof e.data === "string") {
    console.log("Worker: " + e.data);
    return;
}
```

Благодаря этому дополнительному программному коду фоновый поток получает возможность отображать отладочные сообщения, просто передавая строки методу `postMessage()`.

Пример 22.8. Выполнение синхронных HTTP-запросов в фоновом потоке

```
// Этот файл будет загружен вызовом конструктора Worker(), поэтому он будет выполняться
// в независимом потоке выполнения и может безопасно использовать синхронный прикладной
// интерфейс объекта XMLHttpRequest. В качестве сообщения фоновому потоку должен
// передаваться массив URL-адресов. Поток синхронно извлечет содержимое
// из указанных адресов и вернет его в виде массива строк.
onmessage = function(e) {
    var urls = e.data; // Входные данные: URL-адреса извлекаемого содержимого
    var contents = []; // Выходные данные: содержимое указанных URL-адресов

    for(var i = 0; i < urls.length; i++) {
        var url = urls[i];                  // Для каждого URL-адреса
        var xhr = new XMLHttpRequest();  // Создать HTTP-запрос
        xhr.open("GET", url, false);     // false обеспечит синхронное выполн.
        xhr.send();                         // Блокируется до выполнения запроса
        if (xhr.status !== 200)             // Возбудить исключение при неудаче
            throw Error(xhr.status + " " + xhr.statusText + ": " + url);
        contents.push(xhr.responseText); // Иначе сохранить содержимое
    }

    // Отослать массив содержимого URL-адресов обратно основному потоку
    postMessage(contents);
}
```

22.5. Типизированные массивы и буферы

Как говорилось в главе 7, массивы в языке JavaScript являются многоцелевыми объектами с числовыми именами свойств и специальным свойством `length`. Элементами массива могут быть любые JavaScript-значения. Массивы могут увеличиваться и уменьшаться в размерах и быть разреженными. Реализации JavaScript выполняют множество оптимизаций, благодаря которым типичные операции с массивами в языке JavaScript выполняются очень быстро. *Типизированные массивы* – это объекты, подобные массивам (раздел 7.11), которые имеют несколько важных отличий от обычных массивов:

- Все элементы типизированного массива являются числами. Конструктор, используемый для создания массива, определяет тип (целые числа со знаком или без знака или вещественные числа) и размер (в битах) чисел.
- Типизированные массивы имеют фиксированную длину.
- Элементы типизированного массива всегда инициализируются значением 0 при его создании.

Всего существует восемь разновидностей типизированных массивов, каждый с различным типом элементов. Создавать их можно с помощью следующих конструкторов:

Конструктор	Числовой тип
`Int8Array()`	байты со знаком
`Uint8Array()`	байты без знака
`Int16Array()`	16-битные короткие целые со знаком
`Uint16Array()`	16-битные короткие целые без знака
`Int32Array()`	32-битные целые со знаком
`Uint32Array()`	32-битные целые без знака
`Float32Array()`	32-битные вещественные значения
`Float64Array()`	64-битные вещественные значения

При создании типизированного массива конструктору передается размер массива. Вместо размера можно передать массив или типизированный массив, который будет использован для инициализации элементов нового массива. После создания типизированного массива его элементы можно читать или изменять с помощью обычной формы записи с квадратными скобками, как и при работе с любым другим объектом, подобным массиву:

```
var bytes = new Uint8Array(1024);     // Один килобайт байтов
for(var i = 0; i < bytes.length; i++) // Для каждого элемента массива
    bytes[i] = i & 0xFF;              // Записать 8 младших бит индекса
var copy = new Uint8Array(bytes);     // Создать копию массива
var ints = new Int32Array([0,1,2,3]); // Типизированный массив с 4 целыми
```

Современные реализации JavaScript оптимизируют операции с массивами и делают их очень эффективными. Однако типизированные массивы могут быть еще более эффективными, как по времени выполнения операций с ними, так и по использованию памяти. Следующая функция вычисляет наибольшее простое число, которое меньше указанного значения. Она использует алгоритм «Решето Эратосфена», который основан на использовании большого массива, в котором запоминается, какие числа являются простыми, а какие составными. Так как в каждом элементе массива требуется сохранять всего один бит информации, объект `Int8Array` может оказаться более эффективным в использовании, чем обычный массив JavaScript:

```
// Возвращает наибольшее целое простое число меньше n.
// Использует алгоритм "Решето Эратосфена"
function sieve(n) {
    var a = new Int8Array(n+1);   // в a[x] записывается 1, если x - составное число
    var max = Math.floor(Math.sqrt(n)); // Множитель не может быть выше этого значения
    var p = 2;                          // 2 - первое простое число
    while(p <= max) {                   // Для простых чисел меньше max
        for(var i = 2*p; i <= n; i += p) // Пометить числа, кратные p,
            a[i] = 1;                    // как составные
        while(a[++p]) /* пустое тело */; // Следующий непомеченный индекс -
    }                                    // простое число
    while(a[n]) n--; // Цикл в обр. напр., чтобы отыскать последнее простое
    return n;        // И вернуть его
}
```

Функция `sieve()` будет работать, если вызов конструктора `Int8Array()` заменить вызовом традиционного конструктора `Array()`, но выполняться она будет в два-три раза дольше и будет расходовать гораздо больше памяти при больших значениях параметра `n`. Типизированные массивы могут также пригодиться при обработке графических изображений и для математических вычислений:

```
var matrix = new Float64Array(9); // Матрица 3x3
var 3dPoint = new Int16Array(3);  // Точка в 3-мерном пространстве
var rgba = new Uint8Array(4);     // 4-байтовое значение RGBA пиксела
var sudoku = new Uint8Array(81);  // Доска 9x9 для игры в судоку
```

Форма записи с квадратными скобками, используемая в языке JavaScript, позволяет читать и записывать значения отдельных элементов типизированного массива. Но типизированные массивы определяют дополнительные методы для записи и чтения целого фрагмента массива. Метод `set()` копирует элементы обычного или типизированного массива в типизированный массив:

```
var bytes = new Uint8Array(1024)         // Буфер размером 1Кбайт
var pattern = new Uint8Array([0,1,2,3]); // Массив из 4 байтов
bytes.set(pattern);       // Скопировать их в начало другого массива байтов
bytes.set(pattern, 4);   // Скопировать их же в другое место массива
bytes.set([0,1,2,3], 8); // Просто скопировать значения из обычного массива
```

Типизированные массивы имеют также метод `subarray()`, возвращающий фрагмент массива, относительно которого он был вызван:

```
var ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);       // 10 коротких целых
var last3 = ints.subarray(ints.length-3, ints.length); // Последние 3 из них
last3[0]       // => 7: то же самое, что и ints[7]
```

Обратите внимание, что метод `subarray()` не создает копии данных. Он просто возвращает новое представление тех же самых значений:

```
ints[9] = -1; // Изменить значение в оригинальном массиве и...
last3[2]      // => -1: изменения коснулись фрагмента массива
```

Тот факт, что метод `subarray()` возвращает новое представление существующего массива, раскрывает важную особенность типизированных массивов: все они являются представлениями участка памяти, который называется `ArrayBuffer`. Каждый типизированный массив имеет три свойства, связывающие его с лежащим в его основе буфером:

```
last3.buffer                // => вернет объект ArrayBuffer
last3.buffer == ints.buffer // => true: оба - представления одного буфера
last3.byteOffset // => 14: это представление начинается с 14-го байта в буфере
last3.byteLength // => 6: размер представления 6 байт (3 16-битных целых)
```

Сам объект `ArrayBuffer` имеет только одно свойство, возвращающее его длину:

```
last3.byteLength            // => 6: размер представления 6 байт
last3.buffer.byteLength     // => 20: но буфер имеет размер 20 байт
```

Типизированные массивы, элемент <canvas> и базовый JavaScript

Типизированные массивы являются важной частью прикладного интерфейса создания трехмерной графики WebGL в элементе `<canvas>`, и в броузерах они реализованы как часть прикладного интерфейса WebGL. WebGL не рассматривается в этой книге, но типизированные массивы весьма полезны сами по себе, и поэтому обсуждаются здесь. В главе 21 говорилось, что прикладной интерфейс объекта `Canvas` определяет метод `getImageData()`, возвращающий объект `ImageData`. Свойство `data` объекта `ImageData` является массивом байтов. В спецификации HTML он называется `CanvasPixelArray`, но, по сути, это то же самое, что описываемый здесь `Uint8Array`, за исключением способа обработки значений, выходящих за диапазон 0–255.

Имейте в виду, что эти типы не являются частью базового языка. Будущие версии языка JavaScript, возможно, будут включать поддержку типизированных массивов, подобных этим, но на момент написания этих строк еще было неясно, примет ли язык прикладной интерфейс, описываемый здесь, или будет создан новый прикладной интерфейс.

Объект ArrayBuffers — это всего лишь последовательность байтов. К этим байтам можно обращаться с помощью типизированных массивов, но сам объект ArrayBuffer не является типизированным массивом. Однако будьте внимательны: объект ArrayBuffer можно индексировать числами, как любой другой объект JavaScript, но это не обеспечивает доступ к байтам в буфере:

```
var bytes = new Uint8Array(8); // Разместить 8 байтов
bytes[0] = 1;                   // Записать в первый байт значение 1
bytes.buffer[0]        // => undefined: буфер не имеет индекса 0
bytes.buffer[1] = 255; // Попробовать некорректно записать значение в байт буфера
bytes.buffer[1]        // => 255: это обычное JavaScript-свойство
bytes[1]               // => 0: строка выше не изменила байт
```

Имеется возможность создавать объекты ArrayBuffer непосредственно, вызовом конструктора ArrayBuffer(), и на основе имеющегося объекта ArrayBuffer можно создать любое число представлений типизированного массива:

```
var buf = new ArrayBuffer(1024*1024);        // Один Мбайт
var asbytes = new Uint8Array(buf);           // Представление в виде байтов
var asints = new Int32Array(buf);            // В виде 32-битных целых со знаком
var lastK = new Uint8Array(buf,1023*1024);   // Последний Кбайт в виде байтов
var ints2 = new Int32Array(buf, 1024, 256); // 2-й Кбайт в виде 256 целых чисел
```

Типизированные массивы позволяют представлять одну и ту же последовательность байтов в виде целых чисел размером 8, 16, 32 или 64 бита. Это поднимает проблему «порядка следования байтов», т. е. порядка, в каком следуют байты при объединении в более длинные слова. Для эффективности типизированные массивы используют порядок следования байтов, определяемый аппаратным обеспечением. В системах с обратным порядком следования байтов байты числа располагаются в буфере ArrayBuffer в порядке от младшего к старшему. На платформах с прямым порядком следования байтов байты располагаются в порядке от старшего к младшему. Определить порядок следования байтов на текущей платформе, где выполняется сценарий, можно следующим образом:

```
// Если целое число 0x00000001 располагается в памяти в виде
// последовательности байтов 01 00 00 00, следовательно, сценарий выполняется
// на платформе с обратным порядком следования байтов. На платформе с прямым
// порядком следования байтов мы получим байты 00 00 00 01.
var little_endian = new Int8Array(new Int32Array([1]).buffer)[0] === 1;
```

В настоящее время наибольшее распространение получили процессорные архитектуры с обратным порядком следования байтов. Однако многие сетевые протоколы и некоторые двоичные форматы файлов требуют, чтобы байты следовали в прямом порядке. В разделе 22.6 вы узнаете, как использовать объекты ArrayBuffer для хранения байтов, прочитанных из файлов или полученных из сети. В подобных ситуациях нельзя просто полагаться на то, что порядок следования байтов, поддерживаемый аппаратной частью, совпадает с порядком следования байтов в данных. Вообще, при работе с внешними данными, для представления данных в виде массива отдельных байтов можно использовать Int8Array и Uint8Array, но нельзя использовать другие виды типизированных массивов для представления данных в виде массивов многобайтовых слов. Вместо этого можно использовать класс DataView, который определяет методы чтения и записи значений из буфера ArrayBuffer, использующие явно указанный порядок следования байтов:

```
var data; // Предположим, что данные в ArrayBuffer получены из сети
var view = DataView(data);     // Создать представление буфера
var int = view.getInt32(0);    // 32-битное целое со знаком с прямым порядком
                               // следования байтов, начиная с 0-го байта
int = view.getInt32(4,false);  // Следующее 32-битное целое, также с прямым
                               // порядком следования байтов
int = view.getInt32(8,true)    // Следующие 4 байта как целое со знаком
                               // и с обратным порядком следования байтов
view.setInt32(8,int,false);    // Записать его обратно, в формате с прямым
                               // порядком следования байтов
```

Класс `DataView` определяет восемь методов `get` для каждого из восьми видов типизированных массивов. Они имеют такие имена, как `getInt16()`, `getUint32()` и `getFloat64()`. В первом аргументе они принимают смещение значения в байтах в буфере `ArrayBuffer`. Все эти методы чтения, кроме `getInt8()` и `getUint8()`, принимают логическое значение во втором необязательном аргументе. Если второй аргумент отсутствует или имеет значение `false`, используется прямой порядок следования байтов. Если второй аргумент имеет значение `true`, используется обратный порядок следования байтов.

Класс `DataView` определяет восемь соответствующих методов `set`, которые записывают значения в буфер `ArrayBuffer`. В первом аргументе этим методам передается смещение начала значения. Во втором аргументе – записываемое значение. Все методы, кроме `setInt8()` и `setUint8()`, принимают необязательный третий аргумент. Если аргумент отсутствует или имеет значение `false`, значение записывается в формате с прямым порядком следования байтов, когда первым следует старший байт. Если аргумент имеет значение `true`, значение записывается в формате с обратным порядком следования байтов, когда первым записывается младший байт.

22.6. Двоичные объекты

Двоичный объект (`Blob`) – это нетипизированная ссылка, или дескриптор, блока данных. Название «Blob» пришло из мира баз данных SQL, где оно расшифровывается как «Binary Large Object» (большой двоичный объект). В языке JavaScript двоичные объекты часто представляют двоичные данные, и они могут иметь большой размер, но это совсем необязательно: двоичный объект `Blob` может также представлять содержимое небольшого текстового файла. Двоичные объекты непрозрачны, т. е. являются своего рода черными ящиками: все, что можно с ними сделать, – это определить их размер в байтах, узнать MIME-тип и разбить на более мелкие двоичные объекты:

```
var blob = ... // Как получить двоичный объект, будет показано ниже
blob.size      // Размер двоичного объекта в байтах
blob.type      // MIME-тип двоичного объекта или "", если неизвестен
var subblob = blob.slice(0,1024, "text/plain"); // Первый килобайт - как текст
var last = blob.slice(blob.size-1024, 1024);    // Последний килобайт -
                                                // как нетипизированные данные
```

Веб-броузеры могут сохранять двоичные объекты в памяти или на диске, и двоичные объекты могут представлять действительно огромные блоки данных (такие как видеофайлы), которые слишком велики, чтобы их можно было уместить в памяти, предварительно не разбив на более мелкие части с помощью метода `slice()`. Поскольку двоичные объекты могут иметь огромный размер и для работы с ними может потребоваться доступ к диску, прикладные интерфейсы для работы с ними действуют асинхронно (имеются также синхронные версии для использования в фоновых потоках выполнения).

Сами по себе двоичные объекты не представляют особого интереса, но они служат важным механизмом обмена данными для некоторых прикладных интерфейсов в языке JavaScript, которые работают с двоичными данными. На рис. 22.2 показано, как можно обмениваться двоичными объектами во Всемирной паутине, читать и сохранять их в локальной файловой системе, в локальных базах данных и даже обмениваться ими с другими окнами и фоновыми потоками выполнения. Он также показывает, как можно получить содержимое двоичного объекта в виде текста, типизированного массива или URL-адреса.

Прежде чем приступать к работе с двоичным объектом, его необходимо получить. Сделать это можно множеством способов, одни из которых связаны с использованием уже знакомых вам прикладных интерфейсов, а другие – с прикладными интерфейсами, которые описываются ниже в этой главе:

- Двоичные объекты поддерживаются алгоритмом структурированного копирования (смотрите врезку «Структурированные копии» выше), а это означает, что их можно получить от другого окна или фонового потока выполнения вместе с событием «message» (разделы 22.3 и 22.4).

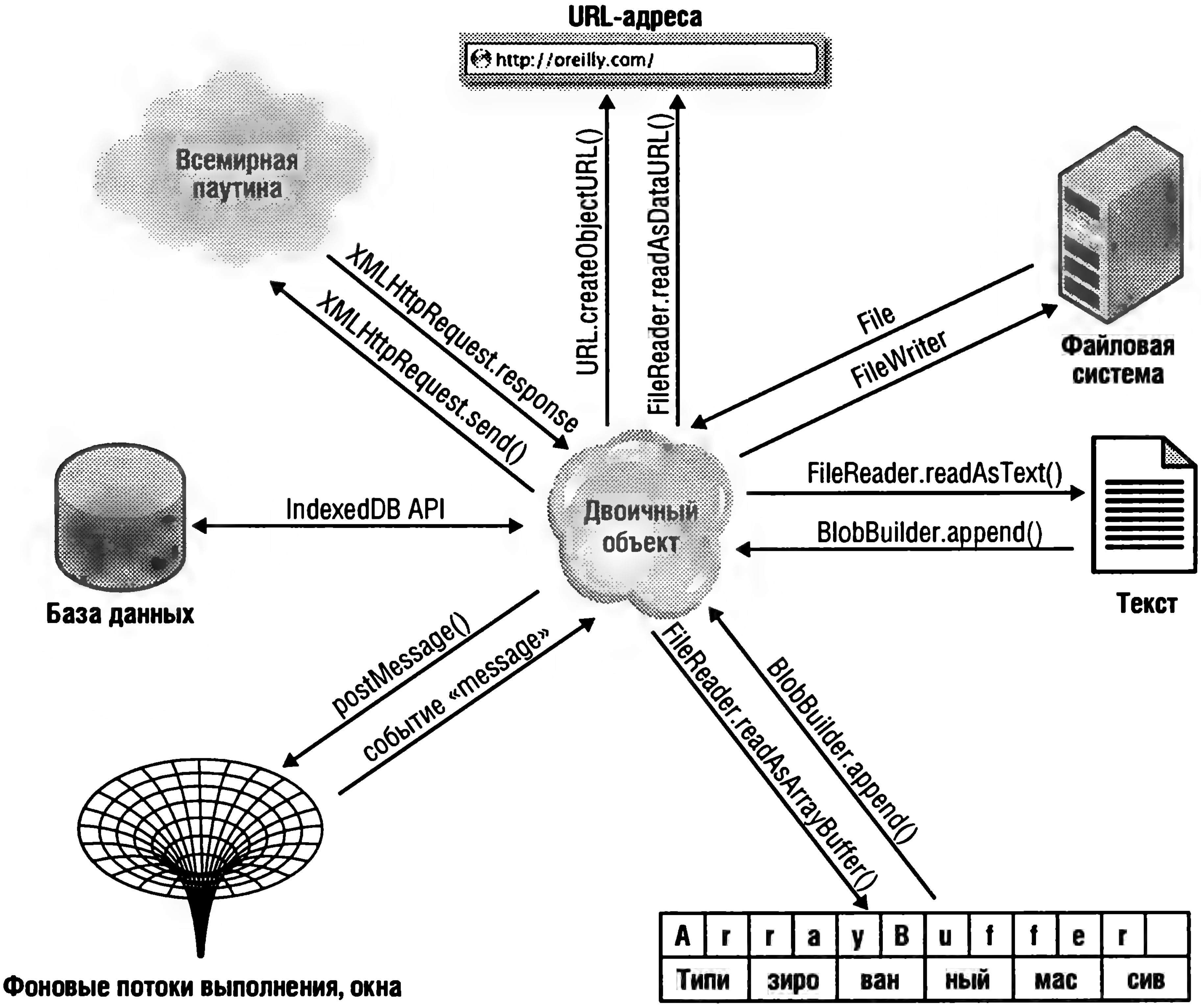

URL-адреса
http://oreilly.com/
Всемирная паутина
URL.createObjectURL()
FileReader.readAsDataURL()
XMLHttpRequest.response
XMLHttpRequest.send()
File
FileWriter
Файловая система
FileReader.readAsText()
BlobBuilder.append()
Текст
IndexedDB API
Двоичный объект
База данных
postMessage()
событие «message»
FileReader.readAsArrayBuffer()
BlobBuilder.append()
Фоновые потоки выполнения, окна
A r r a y B u f f e r
Типи зиро ван ный мас сив

Рис. 22.2. Двоичные объекты и прикладные интерфейсы, использующие их

- Двоичные объекты можно извлекать из баз данных на стороне клиента, как описывается в разделе 22.8.
- Двоичные объекты можно загрузить из сети по протоколу HTTP, используя ультрасовременные возможности, определяемые спецификацией XHR2. Об этом рассказывается в разделе 22.6.2.
- Можно также создать свой двоичный объект, сконструировав его с помощью объекта `BlobBuilder` из строки, объекта `ArrayBuffer` (раздел 22.5) или другого двоичного объекта. Объект `BlobBuilder` будет демонстрироваться в разделе 22.6.3.
- Наконец, и, пожалуй, самое важное, – объект `File` в клиентском JavaScript является подтипом типа `Blob`: объект `File` – это просто двоичный объект с данными, которые имеют имя и дату последнего изменения. Получить объект `File` можно из элемента `<input type="file">` и от прикладного интерфейса буксировки мышью (drag-and-drop), как описывается в разделе 22.6.1. Объект `File` можно также получить с помощью прикладного интерфейса доступа к файловой системе, который охватывается в разделе 22.7.

Получив двоичный объект, над ним можно выполнить различные операции, многие из которых являются симметричными операциям, описанным выше:

- Двоичный объект можно отправить другому окну или фоновому потоку выполнения с помощью метода `postMessage()` (разделы 22.3 и 22.4).
- Двоичный объект можно сохранить в базе данных на стороне клиента (раздел 22.8).
- Двоичный объект можно выгрузить на сервер, передав его методу `send()` объекта `XMLHttpRequest`. Как это можно реализовать, демонстрирует пример 18.9 выгрузки файла (напомню, что объект `File` – это всего лишь специализированная разновидность двоичного объекта `Blob`).
- Можно воспользоваться функцией `createObjectURL()`, чтобы получить специальный URL-адрес вида `blob://`, ссылающийся на двоичное содержимое, и затем использовать его вместе с механизмами DOM или CSS. Этот прием демонстрируется в разделе 22.6.4.
- Можно воспользоваться объектом `FileReader`, чтобы асинхронно (или синхронно, в фоновом потоке выполнения) извлечь содержимое двоичного объекта в строку или в объект `ArrayBuffer`. Этот прием демонстрируется в разделе 22.6.5.
- Можно воспользоваться прикладным интерфейсом доступа к файловой системе и объектом `FileWriter`, который описывается в разделе 22.7, чтобы записать двоичный объект в локальный файл.

В следующих подразделах демонстрируются простые способы получения и использования двоичных объектов. Более сложные приемы, связанные с использованием локальной файловой системы и базами данных на стороне клиента, будут описаны далее, в отдельных разделах.

22.6.1. Файлы как двоичные объекты

Элемент `<input type="file">` изначально предназначался для обеспечения возможности выгрузки файлов в HTML-формах. Производители броузеров всегда с особым тщанием подходили к реализации этого элемента, чтобы позволить ему выгружать только те файлы, которые были явно выбраны пользователем. Сценарии не смогут присвоить имя файла свойству `value` этого элемента, поэтому они лишены возможности выгружать произвольные файлы, находящиеся на компьютере пользователя. Совсем недавно производители броузеров расширили возможности этого элемента с целью обеспечить доступ к файлам на стороне клиента, выбранным пользователем. Обратите внимание, что возможность читать содержимое выбранных пользователем файлов клиентскими сценариями не более и не менее опасна, чем возможность выгружать эти файлы на сервер.

В броузерах, поддерживающих доступ к локальным файлам, свойство `files` элемента `<input type="file">` будет ссылаться на объект `FileList`. Это объект, подобный массиву, элементами которого являются объекты `File`, соответствующие файлам, выбранным пользователем. Объект `File` – это двоичный объект `Blob`, который имеет дополнительные свойства `name` и `lastModifiedDate`:

```
<script>
// Выводит информацию о выбранных файлах
function fileinfo(files) {
    for(var i = 0; i < files.length; i++) { // files - подобный массиву объект
        var f = files[i];
        console.log(f.name,                  // Только имя: без пути к файлу
                    f.size, f.type,          // размер и тип - свойства Blob
                    f.lastModifiedDate);     // еще одно свойство объекта File
    }
}
</script>
<!-- Разрешить выбор нескольких файлов изображений и передать их fileinfo()-->
<input type="file" accept="image/*" multiple onchange="fileinfo(this.files)"/>
```

Возможность выводить имена, типы и размеры файлов не представляет особого интереса. В разделах 22.6.4 и 22.6.5 будет показано, как можно использовать содержимое файла.

В дополнение к файлам, выбранным с помощью элемента `<input>`, пользователь может также дать сценарию доступ к локальным файлам, буксируя их мышью и сбрасывая в окно броузера. Когда приложение получает событие «drop», свойство `dataTransfer.files` объекта события будет содержать ссылку на объект `FileList`, связанный с этой операцией буксировки, если в ней участвовали файлы. Прикладной интерфейс буксировки объектов мышью рассматривался в разделе 17.7, а подобное использование файлов демонстрируется в примере 22.10.

22.6.2. Загрузка двоичных объектов

Глава 18 охватывает тему выполнения HTTP-запросов с помощью объекта `XMLHttpRequest` и также описывает некоторые новые возможности, определяемые проектом спецификации «XMLHttpRequest Level 2» (XHR2). На момент написания этих строк спецификация XHR2 определяла способ загрузки содержимого URL-адреса в виде двоичного объекта, но реализации броузеров пока не поддерживали его. Поскольку программный код не может быть протестирован, этот раздел является лишь схематическим описанием прикладного интерфейса, предусматриваемого спецификацией XHR2 для работы с двоичными объектами.

Пример 22.9 демонстрирует простой способ загрузки двоичного объекта из Веб. Сравните его с примером 18.2, который загружает содержимое URL-адреса как простой текст.

Пример 22.9. Загрузка двоичного объекта с помощью объекта XMLHttpRequest

```
// Запрашивает методом GET содержимое URL в виде двоичного объекта и передает его
// указанной функции обратного вызова. Этот программный код не тестировался: на тот
// момент, когда он был написан, броузеры еще не поддерживали этот прикладной интерфейс.
function getBlob(url, callback) {
    var xhr = new XMLHttpRequest(); // Создать новый объект XHR
    xhr.open("GET", url);           // Указать URL-адрес
    xhr.responseType = "blob"       // Желательно получить двоичный объект
    xhr.onload = function() {       // onload проще, чем onreadystatechange
        callback(xhr.response);     // Передать ответ функции обратного вызова
    }                               // Отметьте: .response, а не .responseText
    xhr.send(null);                 // Послать запрос
}
```

Если загружаемый двоичный объект слишком велик и вам хотелось бы начать его обработку уже в процессе загрузки, можно задействовать обработчик события onprogress **в комплексе с приемами чтения двоичных объектов, которые демонстрируются в разделе 22.6.5.**

22.6.3. Конструирование двоичных объектов

Двоичные объекты часто представляют фрагменты данных из внешних ресурсов, таких как локальные файлы, URL-адреса или базы данных. Но иногда веб-приложению требуется создать собственный двоичный объект, чтобы выгрузить его на веб-сервер, сохранить в файле или в базе данных, или передать его фоновому потоку выполнения. Создать объект Blob **из имеющихся данных можно с помощью объекта** BlobBuilder**:**

```
// Создать новый объект BlobBuilder
var bb = new BlobBuilder();
// Добавить в двоичный объект строку и отметить ее конец символом NUL
bb.append("Данный двоичный объект содержит этот текст и 10 " +
          "32-битных целых чисел с прямым порядком следования байтов.");
bb.append("\0"); // Добавить символ NUL, чтобы отметить конец строки
// Сохранить некоторые данные в объекте ArrayBuffer
var ab = new ArrayBuffer(4*10);
var dv = new DataView(ab);
for(var i = 0; i < 10; i++) dv.setInt32(i*4,i);
// Добавить ArrayBuffer в двоичный объект
bb.append(ab);
// Теперь извлечь полученный двоичный объект, указав искусственный MIME-тип
var blob = bb.getBlob("x-optional/mime-type-here");
```

В начале этого раздела мы узнали, что двоичные объекты имеют метод slice()**, который разбивает их на фрагменты. Точно так же имеется возможность объединять двоичные объекты, передавая их методу** append() **объекта** BlobBuilder**.**

22.6.4. URL-адреса, ссылающиеся на двоичные объекты

В предыдущих разделах было показано, как можно получить или создать двоичный объект. Теперь мы поговорим о том, что можно *делать* с полученными или созданными двоичными объектами. Самое простое, что можно сделать с двоичным объектом, – это создать URL-адрес, ссылающийся на него. Такой URL-адрес

можно использовать везде, где используются обычные URL-адреса: в элементах DOM, в таблицах стилей и даже при работе с объектом `XMLHttpRequest`.

Создаются URL-адреса, ссылающиеся на двоичные объекты, с помощью функции `createObjectURL()`. На момент написания этих строк проект спецификации и Firefox 4 помещали эту функцию в глобальный объект `URL`, а броузер Chrome и библиотека Webkit добавляли свой префикс к имени этого объекта, называя его `webkitURL`. Ранние версии спецификации (и ранние реализации броузеров) помещали эту функцию непосредственно в объект `Window`. Чтобы получить возможность переносимым образом создавать URL-адреса, ссылающиеся на двоичные объекты, можно определить вспомогательную функцию, как показано ниже:

```
var getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||
    (window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||
    window.createObjectURL;
```

Фоновые потоки выполнения также могут использовать этот прикладной интерфейс и обращаться к тем же функциям, в том же объекте `URL` (или `webkitURL`).

Если передать двоичный объект функции `createObjectURL()`, она вернет URL-адрес (в виде обычной строки). Этот адрес начинается с названия схемы `blob://`, за которой следует короткая строка, ссылающаяся на двоичный объект с некоторым уникальным идентификатором. Обратите внимание, что эти URL-адреса совершенно не похожи на URL-адреса `data://`, которые представляют свое собственное содержимое. URL-адрес, ссылающийся на двоичный объект, – это просто ссылка на объект `Blob`, хранящийся в памяти броузера или на диске. URL-адреса вида `blob://` также отличаются от URL-адресов `file://`, которые ссылаются непосредственно на файл в локальной файловой системе, давая возможность увидеть путь к файлу, позволяя просматривать содержимое каталогов и тем самым затрагивая проблемы безопасности.

Пример 22.10 демонстрирует два важных приема. Во-первых, он реализует «площадку для сброса», которая обрабатывает события механизма буксировки мышью (drag-and-drop), имеющие отношение к файлам. Во-вторых, когда пользователь сбросит один или более файлов на эту «площадку», с помощью функции `createObjectURL()` для каждого из файлов будет создан URL-адрес и элемент `<img>`, чтобы отобразить миниатюры изображений, на которые ссылаются эти URL-адреса.

Пример 22.10. Отображение файлов изображений с использованием URL-адресов двоичных объектов

```
<!DOCTYPE html>
<html><head>
<script>
// На момент написания этих строк создатели Firefox и Webkit еще не пришли
// к соглашению об именовании функции createObjectURL()
var getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||
    (window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||
    window.createObjectURL;
var revokeBlobURL = (window.URL && URL.revokeObjectURL.bind(URL)) ||
    (window.webkitURL && webkitURL.revokeObjectURL.bind(webkitURL)) ||
    window.revokeObjectURL;

// После загрузки документа добавить обработчики событий к элементу droptarget,
// чтобы он мог обрабатывать сбрасываемые файлы
window.onload = function() {
    // Отыскать элемент, к которому следует добавить обработчики событий.
    var droptarget = document.getElementById("droptarget");

    // Выделяет элемент droptarget изменением рамки, когда пользователь
    // буксирует файлы над ним.
    droptarget.ondragenter = function(e) {
        // Игнорировать, если буксируется что-то иное, не являющееся файлом.
        // Когда будет реализована поддержка атрибута dropzone, определяемого
        // спецификацией HTML5, это позволит упростить обработчик.
        var types = e.dataTransfer.types;
        if (!types ||
            (types.contains && types.contains("Files")) ||
            (types.indexOf && types.indexOf("Files") != -1)) {
            droptarget.classList.add("active"); // Выделить элемент droptarget
            return false;                       // Нас интересует
        }                                       // буксируемый объект
    };
    // Снимает выделение площадки сброса, как только пользователь
    // отбуксирует файл за ее пределы
    droptarget.ondragleave = function() {
        droptarget.classList.remove("active");
    };

    // Этот обработчик просто сообщает броузеру продолжать посылать события
    droptarget.ondragover = function(e) { return false; };

    // Когда пользователь сбросит файлы, необходимо получить их URL-адреса
    // и отобразить миниатюры.
    droptarget.ondrop = function(e) {
        var files = e.dataTransfer.files;            // Сброшенные файлы
        for(var i = 0; i < files.length; i++) {      // Обойти все файлы в цикле
            var type = files[i].type;
            if (type.substring(0,6) !== "image/")    // Пропустить не являющиеся
                continue;                            // изображениями
            var img = document.createElement("img"); // Создать элемент <img>
            img.src = getBlobURL(files[i]);          // URL blob:// в <img>
            img.onload = function() {                // После загрузки изобр.
                this.width = 100;                    // установить его размеры
                document.body.appendChild(this);     // и вставить в документ.
                revokeBlobURL(this.src);             // Предотвратить утечки памяти!
            }
        }

        droptarget.classList.remove("active");       // Снять выделение
        return false;                                // Событие сброса обработано
    }
};
</script>
<style> /* Простые стили для оформления площадки сброса */
#droptarget { border: solid black 2px; width: 200px; height: 200px; }
#droptarget.active { border: solid red 4px; }
</style>
</head>
<body> <!-- Изначально в документе имеется только площадка сброса -->
<div id="droptarget">Сбросьте сюда файлы изображений</div>
</body>
</html>
```

URL-адреса двоичных объектов имеют то же происхождение (раздел 13.6.2), что и сценарии, создавшие их. Это делает их более универсальными по сравнению с URL-адресами `file://`, которые имеют иное происхождение, из-за чего последние сложнее использовать в веб-приложениях. URL-адреса двоичных объектов считаются допустимыми только в документах с общим происхождением. Если, например, с помощью метода `postMessage()` передать URL-адрес двоичного объекта в окно с документом, имеющим другое происхождение, для этого окна URL-адрес будет бессмысленным.

URL-адреса двоичных объектов не являются постоянными. Такой URL-адрес перестанет быть действительным, как только пользователь закроет документ или выйдет из документа, в котором был создан этот URL-адрес. Нельзя, например, сохранить URL-адрес двоичного объекта в локальном хранилище и затем повторно использовать его, когда пользователь начнет новый сеанс работы с веб-приложением.

Имеется также возможность вручную «прекращать» действие URL-адреса двоичного объекта вызовом метода `URL.revokeObjectURL()` (или `webkitURL.revokeObjectURL()`), и, как вы могли заметить, пример 22.10 использует эту возможность. Это связано с проблемой управления памятью. После того как миниатюра будет отображена, двоичный объект становится ненужным и его следует сделать доступным для утилизации сборщиком мусора. Но если веб-броузер будет хранить ссылку на созданный двоичный объект в виде URL-адреса двоичного объекта, он не сможет утилизировать его, даже если он не будет больше использоваться в приложении. Интерпретатор JavaScript не может следить за использованием строк, и если URL-адрес по-прежнему остается допустимым, он вправе предположить, что этот адрес все еще используется. Это означает, что интерпретатор не сможет утилизировать двоичный объект, пока не будет прекращено действие URL-адреса. Пример 22.10 работает с локальными файлами, не требующими утилизации, но представьте, какие серьезные утечки памяти могут быть при работе с двоичными объектами, создаваемыми в памяти методом `BlobBuilder` или загружаемыми с помощью объекта `XMLHttpRequest` и сохраняемыми во временных файлах.

URL-схема `blob://` явно проектировалась как упрощенный вариант схемы `http://`, и при обращении по URL-адресу `blob://` броузеры должны действовать как своеобразные HTTP-серверы. При запросе недействительного URL-адреса двоичного объекта броузер должен послать в ответ код состояния 404 «Not Found». При запросе URL-адреса двоичного объекта с другим происхождением броузер должен вернуть код состояния 403 «Not Allowed». URL-адреса двоичных объектов могут использоваться только в запросах GET, и в случае успешного выполнения запроса броузер должен отправить код состояния 200 «OK» и заголовок `Content-Type` со значением свойства `type` двоичного объекта `Blob`. Поскольку URL-адреса двоичных объектов действуют как упрощенные URL-адреса `http://`, их содержимое можно «загружать» с помощью объекта `XMLHttpRequest`. (Однако, как будет показано в следующем разделе, содержимое двоичного объекта можно прочитать более непосредственным способом – с помощью объекта `FileReader`.)

22.6.5. Чтение двоичных объектов

До сих пор двоичные объекты были для нас непрозрачными фрагментами данных, которые позволяют обращаться к их содержимому только косвенным способом, посредством URL-адресов двоичных объектов. Объект `FileReader` дает возможность

читать символы или байты, хранящиеся в двоичном объекте, и его можно рассматривать как противоположность объекту `BlobBuilder`. (Для него больше подошло бы имя `BlobReader`, поскольку он может работать с любыми двоичными объектами, а не только с файлами.) Так как двоичные объекты могут быть очень большими и храниться в файловой системе, прикладной интерфейс чтения их содержимого действует асинхронно, во многом подобно тому, как действует объект `XMLHttpRequest`. Фоновым потокам доступна также синхронная версия прикладного интерфейса в виде объекта `FileReaderSync`, хотя они могут использовать и асинхронную версию.

Чтобы воспользоваться объектом `FileReader`, сначала необходимо создать его экземпляр с помощью конструктора `FileReader()`. Затем определить обработчики событий. Обычно в приложениях определяются обработчики событий «load» и «error» и иногда – обработчик событий «progress». Сделать это можно посредством свойств `onload`, `onerror` и `onprogress` или с помощью стандартного метода `addEventListener()`. Кроме того, объекты `FileReader` генерируют события «loadstart», «loadend» и «abort», которые соответствуют одноименным событиям в объекте `XMLHttpRequest` (раздел 18.1.4).

После создания объекта `FileReader` и регистрации необходимых обработчиков событий можно передать двоичный объект, содержимое которого требуется прочитать, одному из четырех методов: `readAsText()`, `readAsArrayBuffer()`, `readAsDataURL()` и `readAsBinaryString()`. (Разумеется, можно сначала вызвать один из этих методов и лишь потом зарегистрировать обработчики событий – благодаря однопоточной природе JavaScript, о которой рассказывалось в разделе 22.4, обработчики событий не могут быть вызваны, пока ваша функция не вернет управление и броузер не сможет продолжить цикл обработки событий.) Первые два метода являются наиболее важными, и только они будут описаны здесь. Каждый из этих методов чтения принимает двоичный объект `Blob` в первом аргументе. Метод `readAsText()` принимает необязательный второй аргумент, определяющий имя кодировки текста. Если кодировка не указана, метод автоматически будет обрабатывать текст в кодировках ASCII и UTF-8 (а также текст в кодировке UTF-16 с маркером порядка следования байтов (byte-order mark, BOM)).

По мере чтения содержимого указанного двоичного объекта объект `FileReader` будет обновлять значение своего свойства `readyState`. Первоначально это свойство получает значение 0, показывающее, что еще ничего не было прочитано. Когда становятся доступны какие-нибудь данные, оно принимает значение 1 и по окончании чтения – значение 2. Свойство `result` хранит частично или полностью прочитанные данные в виде строки или объекта `ArrayBuffer`. Веб-приложения обычно не опрашивают свойства `readyState` и `result` и вместо этого используют обработчик события `onprogress` или `onload`.

Пример 22.11 демонстрирует, как использовать метод `readAsText()` для чтения локальных текстовых файлов, выбранных пользователем.

Пример 22.11. Чтение текстовых файлов с помощью объекта FileReader

```
<script>
// Читает указанный текстовый файл и отображает его в элементе <pre> ниже
function readfile(f) {
    var reader = new FileReader(); // Создать объект FileReader
    reader.readAsText(f);          // Прочитать файл
    reader.onload = function() {   // Определить обработчик события
```

```
        var text = reader.result;                      // Это содержимое файла
        var out = document.getElementById("output");   // Найти элемент output
        out.innerHTML = "";                            // Очистить его
        out.appendChild(document.createTextNode(text));// Вывести содержимое
    }                                                  // файла
    reader.onerror = function(e) {                     // Если что-то пошло не так
        console.log("Error", e);                       // Вывести сообщение об ошибке
    };
}
</script>
// Выберите файл для отображения:
<input type="file" onchange="readfile(this.files[0])"></input>
<pre id="output"></pre>
```

Метод `readAsArrayBuffer()` похож на метод `readAsText()`, за исключением того, что требует приложить чуть больше усилий при работе с результатом, возвращая объект `ArrayBuffer` вместо строки. Пример 22.12 демонстрирует использование метода `readAsArrayBuffer()` для чтения первых четырех байтов из файла в виде целого числа с прямым порядком следования байтов.

Пример 22.12. Чтение первых четырех байтов из файла

```
<script>
// Исследует первые 4 байта в указанном двоичном объекте. Если это "сигнатура",
// определяющая тип файла, асинхронно устанавливает свойство двоичного объекта.
function typefile(file) {
    var slice = file.slice(0,4);      // Читать только первые 4 байта
    var reader = new FileReader();    // Создать асинхронный FileReader
    reader.readAsArrayBuffer(slice); // Прочитать фрагмент файла
    reader.onload = function(e) {
        var buffer = reader.result;           // Результат - ArrayBuffer
        var view = new DataView(buffer);      // Получить доступ к результату
        var magic = view.getUint32(0, false); // 4 байта, прямой порядок
        switch(magic) {                       // Определить по ним тип файла
        case 0x89504E47: file.verified_type = "image/png"; break;
        case 0x47494638: file.verified_type = "image/gif"; break;
        case 0x25504446: file.verified_type = "application/pdf"; break;
        case 0x504b0304: file.verified_type = "application/zip"; break;
        }
        console.log(file.name, file.verified_type);
    };
}
</script>
<input type="file" onchange="typefile(this.files[0])"></input>
```

В фоновых потоках выполнения вместо объекта `FileReader` можно использовать объект `FileReaderSync`. Синхронный прикладной интерфейс определяет те же методы `readAsText()` и `readAsArrayBuffer()`, которые принимают те же аргументы, что и их асинхронные версии. Разница заключается лишь в том, что синхронные методы блокируются до окончания операции и непосредственно возвращают результат в вид строки или объекта `ArrayBuffer`, что избавляет от необходимости использовать обработчики событий. Пример 22.14 ниже демонстрирует использование объекта `FileReaderSync`.

22.7. Прикладной интерфейс к файловой системе

В разделе 22.6.5 вы познакомились с классом `FileReader`, использовавшимся для чтения содержимого файлов, выбираемых пользователем, или любых двоичных объектов. Типы `File` и `Blob` определяются проектом спецификации, известной как «File API». Проект другой спецификации, еще более новой, чем «File API», дает веб-приложениям управляемый доступ к частной файловой системе, где они могут писать в файлы, читать файлы, создавать каталоги, читать содержимое каталогов и т. д. На момент написания этих строк данный прикладной интерфейс к файловой системе был реализован только в броузере Google Chrome. Это мощная и важная форма локального хранилища, поэтому она будет описана здесь, несмотря на то что ее прикладной интерфейс еще менее стабилен, чем другие прикладные интерфейсы, описываемые в этой главе. Данный раздел охватывает лишь основные задачи, выполняемые с файловой системой, и не демонстрирует всех возможностей прикладного интерфейса. Так как обсуждаемый здесь прикладной интерфейс является новым и нестабильным, он не описывается в справочном разделе этой книги.

Работа с файлами в локальной файловой системе является многоэтапным процессом. Прежде всего, необходимо получить объект, представляющий саму файловую систему. Сделать это можно с помощью синхронного прикладного интерфейса в фоновом потоке или асинхронного – в основном потоке выполнения:

```
// Метод синхронного получения файловой системы. Принимает параметры,
// определяющие срок существования файловой системы и ее размер.
// Возвращает объект файловой системы или возбуждает исключение.
var fs = requestFileSystemSync(PERSISTENT, 1024*1024);

// Асинхронная версия принимает функции обратного вызова для обработки
// успешного или неудачного создания файловой системы
requestFileSystem(TEMPORARY,     // срок существования
                  50*1024*1024,  // размер: 50 Мбайт
                  function(fs) { // будет вызвана с объектом файловой системы
                      // Здесь используется объект fs
                  },
                  function(e) {       // будет вызвана с объектом ошибки
                      console.log(e); // Или как-то иначе обработать ошибку
                  });
```

В обеих версиях прикладного интерфейса, синхронной и асинхронной, указываются срок существования и желаемый размер файловой системы. Файловая система, срок существования которой определяется константой `PERSISTENT` (постоянная), подходит для веб-приложений, которым требуется хранить пользовательские данные постоянно. Броузер не будет удалять эту файловую систему, пока пользователь явно не потребует этого. Файловая система, срок существования которой определяется константой `TEMPORARY` (временная), подходит для веб-приложений, которые требуют кэшировать данные, но также сохраняют работоспособность и после того, как веб-броузер удалит файловую систему. Размер файловой системы определяется в байтах и должен быть равен разумному верхнему пределу объема данных, которые потребуется сохранять. Броузер может ограничивать этот размер, устанавливая квоты.

Доступность файловой системы определяется происхождением создавшего ее документа. Все документы или веб-приложения с общим происхождением (хост, порт и протокол) будут совместно использовать одну и ту же файловую систему. Два документа или приложения с разным происхождением будут пользоваться совершенно разными и никак не связанными между собой файловыми системами. Кроме того, файловая система веб-приложения отделена от остальных файлов на жестком диске пользователя: веб-приложения не имеют никакой возможности «вырваться» за пределы локального корневого каталога или как-то иначе получить доступ к произвольным файлам.

Обратите внимание, что в именах этих функций присутствует слово «request» (запросить). Когда приложение вызывает одну из этих функций в первый раз, броузер может запросить разрешение у пользователя, прежде чем создать файловую систему и разрешить доступ к ней.[1] После получения доступа все последующие вызовы метода `request` будут просто возвращать объект, представляющий уже имеющуюся локальную файловую систему.

Объект файловой системы, полученный одним из методов, представленных выше, имеет свойство `root`, ссылающееся на корневой каталог файловой системы. Это объект `DirectoryEntry`, и он может иметь вложенные каталоги, представленные собственными объектами `DirectoryEntry`. Каждый каталог в файловой системе может содержать файлы, представленные объектами `FileEntry`. Объект `DirectoryEntry` определяет методы для получения объектов `DirectoryEntry` и `FileEntry` по пути в файловой системе (они могут создавать новые каталоги и файлы, если указанное имя не существует). Объект `DirectoryEntry` также определяет фабричный метод `createReader()`, возвращающий объект `DirectoryReader`, который позволяет получить список содержимого каталога.

Класс `FileEntry` определяет метод для получения объекта `File` (двоичный объект `Blob`), представляющий содержимое файла. Получив этот объект, его содержимое можно прочитать с помощью объекта `FileReader` (как показано в разделе 22.6.5). Объект `FileEntry` определяет еще один метод, возвращающий объект `FileWriter`, который можно использовать для записи в файл.

Операции чтения и записи с использованием этого прикладного интерфейса представляют собой многоэтапный процесс. Прежде всего, необходимо получить объект файловой системы. Затем, используя свойство `root` этого объекта, необходимо отыскать (и при необходимости создать) объект `FileEntry`, представляющий требуемый файл. Затем с помощью объекта `FileEntry` нужно получить объект `File` или `FileWriter` для выполнения операции чтения или записи. Этот процесс становится особенно сложным при использовании асинхронного прикладного интерфейса:

```
// Читает текстовый файл "hello.txt" и выводит его содержимое. При использовании
// асинхронного прикладного интерфейса глубина вложенности функций достигает
// четырех уровней. Этот пример не включает определения функций обработки ошибок.
requestFileSystem(PERSISTENT, 10*1024*1024, function(fs) { // Получить объект ФС
    fs.root.getFile("hello.txt", {}, function(entry) {     // Получить FileEntry
        entry.file(function(file) {                        // Получить File
            var reader = new FileReader();
            reader.readAsText(file);
            reader.onload = function() {                   // Получить содержимое файла
```

---

[1] На момент написания этих строк броузер Chrome не запрашивал разрешение у пользователя, но требовал, чтобы он был запущен с ключом командной строки `--unlimited-quota-for-files`.

```
                console.log(reader.result);
            };
        });
    });
});
```

В примере 22.13 демонстрируется более полное решение. В нем показано, как использовать асинхронный прикладной интерфейс для чтения, записи и удаления файлов, создания каталогов и получения списков их содержимого.

Пример 22.13. Использование асинхронного прикладного интерфейса доступа к файловой системе

```
/*
 * Следующие функции были протестированы в Google Chrome 10.0 dev.
 * Вам может потребоваться запустить Chrome со следующими ключами:
 * --unlimited-quota-for-files : разрешает доступ к файловой системе
 * --allow-file-access-from-files : разрешает тестировать из URL file://
 */

// Многие асинхронные функции, используемые здесь, принимают необязательные функции
// обратного вызова для обработки ошибок.
// Следующая функция просто выводит сообщение об ошибке.
function logerr(e) { console.log(e); }

// requestFileSystem() возвращает локальную файловую систему, доступную
// только приложениям с указанным происхождением. Приложение может читать
// и писать файлы в ней, но не может получить доступ к остальной файловой системе.
var filesystem; // Предполагается, что эта переменная будет инициализирована
                // перед вызовом функции, объявленной ниже.
requestFileSystem(PERSISTENT,    // Или TEMPORARY для кэширования файлов
                  10*1024*1024,  // Требуется 10 Мбайт
                  function(fs) { // После выполнения вызвать эту функцию
                      filesystem = fs; // Просто сохранить ссылку на файловую систему
                  },                   // в глобальной переменной.
                  logerr);             // Вызвать эту функцию в случае ошибки

// Читает содержимое указанного файла как текст и передает его функции обратного вызова.
function readTextFile(path, callback) {
    // Вызвать getFile(), чтобы получить объект FileEntry для файла
    // с указанным именем
    filesystem.root.getFile(path, {}, function(entry) {
        // При вызове этой функции передается объект FileEntry, соответствующий файлу.
        // Теперь следует вызвать метод FileEntry.file(), чтобы получить объект File
        entry.file(function(file) {        // Вызвать с объектом File
            var reader = new FileReader(); // Создать объект FileReader
            reader.readAsText(file);       // И прочитать файл
            reader.onload = function() {   // В случае успешного чтения
                callback(reader.result);   // Передать данные функции callback
            }
            reader.onerror = logerr;   // Сообщить об ошибке в readAsText()
        }, logerr);                    // Сообщить об ошибке в file()
    },
    logerr);                           // Сообщить об ошибке в getFile()
}

// Добавляет указанное содержимое в конец файла с указанным именем, создает новый файл,
// если файл с указанным именем не существует. Вызывает callback по завершении операции.
function appendToFile(path, contents, callback) {
    // filesystem.root - это корневой каталог.
    filesystem.root.getFile( // Получить объект FileEntry
        path,                // Имя и путь к требуемому файлу
        {create:true},       // Создать, если не существует
        function(entry) {    // Вызвать эту функцию, когда файл будет найден
            entry.createWriter(    // Создать для файла объект FileWriter
                function(writer) { // Вызвать эту функцию после создания
                    // По умолчанию запись производится в начало файла.
                    // Нам же требуется выполнить запись в конец файла
                    writer.seek(writer.length);    // Переместиться в конец файла

                    // Преобразовать содержимое файла в объект Blob. Аргумент contents
                    // может быть строкой, объектом Blob или объектом ArrayBuffer.
                    var bb = new BlobBuilder()
                    bb.append(contents);
                    var blob = bb.getBlob();

                    // Записать двоичный объект в файл
                    writer.write(blob);
                    writer.onerror = logerr;       // Сообщить об ошибке в write()
                    if (callback)                  // Если указана функция callback
                        writer.onwrite = callback; // вызвать в случае успеха
                },
                logerr); // Сообщить об ошибке в createWriter()
        },
        logerr);         // Сообщить об ошибке в getFile()
}

// Удаляет файл с указанным именем, вызывает callback по завершении операции.
function deleteFile(name, callback) {
    filesystem.root.getFile(name, {}, // Получить FileEntry по имени файла
                        function(entry) { // Передать FileEntry сюда
                            entry.remove(callback, // Удалить файл
                                         logerr);  // Или сообщить
                        },                         // об ошибке в remove()
                        logerr); // Сообщить об ошибке в getFile()
}

// Создает новый каталог с указанным именем
function makeDirectory(name, callback) {
    filesystem.root.getDirectory(name, // Имя создаваемого каталога
                             {     // Параметры
                                 create: true,  // Создать, если не сущ.
                                 exclusive:true // Ошибка, если существует
                             },
                             callback, // Вызвать по завершении
                             logerr);  // Выводить любые ошибки
}

// Читает содержимое указанного каталога и передает его в виде массива строк
// указанной функции callback
function listFiles(path, callback) {
    // Если каталог не указан, получить содержимое корневого каталога.
    // Иначе отыскать каталог с указанным именем и вернуть список
    // с его содержимым (или сообщить об ошибке поиска).
    if (!path) getFiles(filesystem.root);
```

```
        else filesystem.root.getDirectory(path, {}, getFiles, logerr);

        function getFiles(dir) {             // Эта функция используется выше
            var reader = dir.createReader(); // Объект DirectoryReader
            var list = [];                   // Для сохранения имен файлов
            reader.readEntries(handleEntries, // Передать функции ниже
                               logerr);      // или сообщить об ошибке.

            // Чтение каталогов может превратиться в многоэтапный процесс.
            // Необходимо сохранять результаты вызовов readEntries(), пока не будет
            // получен пустой массив. На этом операция будет закончена,
            // и полный список можно будет передать функции callback.
            function handleEntries(entries) {
                if (entries.length == 0) callback(list); // Операция закончена
                else {
                    // Иначе добавить эти записи в общий список и запросить
                    // очередную порцию. Объект, подобный массиву, содержит
                    // объекты FileEntry, и нам следует получить имя для каждого.
                    for(var i = 0; i < entries.length; i++) {
                        var name = entries[i].name; // Получить имя записи
                        if (entries[i].isDirectory) name += "/"; // Пометить каталоги
                        list.push(name);            // Добавить в список
                    }
                    // Получить следующую порцию записей
                    reader.readEntries(handleEntries, logerr);
                }
            }
        }
    }
```

Работать с файлами и с файловой системой намного проще в фоновых потоках выполнения, где допускается выполнять блокирующие вызовы и можно использовать синхронный прикладной интерфейс. Пример 22.14 определяет те же функции для работы с файловой системой, что и пример 22.13, но использует синхронный прикладной интерфейс, и потому получился намного короче.

Пример 22.14. Синхронный прикладной интерфейс для работы с файловой системой

```
// Утилиты для работы с файловой системой, использующие синхронный прикладной
// интерфейс, предназначенный для фоновых потоков выполнения
var filesystem = requestFileSystemSync(PERSISTENT, 10*1024*1024);

function readTextFile(name) {
    // Получить объект File из объекта FileEntry из корневого DirectoryEntry
    var file = filesystem.root.getFile(name).file();
    // Использовать для чтения синхронную версию FileReader
    return new FileReaderSync().readAsText(file);
}

function appendToFile(name, contents) {
    // Получить FileWriter из FileEntry из корневого DirectoryEntry
    var writer = filesystem.root.getFile(name, {create:true}).createWriter();
    writer.seek(writer.length); // Начать запись с конца файла
    var bb = new BlobBuilder()  // Собрать содержимое в виде объекта Blob
    bb.append(contents);
    writer.write(bb.getBlob()); // Записать двоичный объект в файл
}
```

```
function deleteFile(name) {
    filesystem.root.getFile(name).remove();
}

function makeDirectory(name) {
    filesystem.root.getDirectory(name, { create: true, exclusive:true });
}

function listFiles(path) {
    var dir = filesystem.root;
    if (path) dir = dir.getDirectory(path);

    var lister = dir.createReader();
    var list = [];
    do {
        var entries = lister.readEntries();
        for(var i = 0; i < entries.length; i++) {
            var name = entries[i].name;
            if (entries[i].isDirectory) name += "/";
            list.push(name);
        }
    } while(entries.length > 0);
    return list;
}

// Разрешить основному потоку выполнения использовать эти утилиты, посылая сообщения
onmessage = function(e) {
    // Сообщение должно быть объектом:
    // { function: "appendToFile", args: ["test", "testing, testing"]}
    // Вызвать указанную функцию с указанными аргументами и послать сообщение обратно
    var f = self[e.data.function];
    var result = f.apply(null, e.data.args);
    postMessage(result);
};
```

22.8. Базы данных на стороне клиента

Архитектура веб-приложений традиционно была основана на поддержке HTML, CSS и JavaScript на стороне клиента и базы данных на стороне сервера. Поэтому одним из самых удивительных прикладных интерфейсов, определяемых спецификацией HTML5, является поддержка баз данных на стороне клиента. Это не прикладные интерфейсы доступа к базам данных на стороне сервера, а действительно интерфейсы доступа к базам данных, хранящимся на компьютере клиента и доступным непосредственно из программного кода на языке JavaScript, выполняемого броузером.

Прикладной интерфейс веб-хранилища, определяемый спецификацией «Web Storage» и описанный в разделе 20.1, можно расценивать как простейшую разновидность базы данных, хранящую пары ключ/значение. Но помимо него имеются еще два прикладных интерфейса доступа к клиентским базам данных, которые являются «настоящими» базами данных. Один из них известен как «Web SQL Database» – простая реляционная база данных, поддерживающая простейшие SQL-запросы. Этот прикладной интерфейс реализован в броузерах Chrome, Safari и Opera. Он не реализован в Firefox и IE и, скорее всего, никогда не будет реализован в них. Работа над официальной спецификацией этого прикладного интерфейса была остановлена, и поддержка полноценной базы данных SQL, вероятно, никогда не приобретет статус официального стандарта или неофициальной, но широко поддерживаемой особенности веб-платформы.

В настоящее время все усилия по стандартизации сконцентрированы на другом прикладном интерфейсе к базам данных, известном как IndexedDB. Пока слишком рано описывать детали этого прикладного интерфейса (его описание отсутствует в четвертой части книги), но Firefox 4 и Chrome 11 включают его реализацию, и в этом разделе будет представлен действующий пример, демонстрирующий некоторые из наиболее важных особенностей прикладного интерфейса IndexedDB.

IndexedDB – это объектная, а не реляционная база данных, и она намного проще, чем базы данных, поддерживающие язык запросов SQL. Однако она гораздо мощнее, эффективнее и надежнее, чем веб-хранилище пар ключ/значение, доступное посредством прикладного интерфейса Web Storage. Как и в случае прикладных интерфейсов к веб-хранилищам и файловой системе, доступность базы данных IndexedDB определяется происхождением создавшего ее документа: две веб-страницы с общим происхождением могут обращаться к данным друг друга, но веб-страницы с разным происхождением – нет.

Для каждого происхождения может быть создано произвольное число баз данных IndexedDB. Каждая база данных имеет имя, которое должно быть уникальным для данного происхождения. С точки зрения прикладного интерфейса IndexedDB база данных является простой коллекцией именованных *хранилищ объектов*. Как следует из этого названия, хранилище объектов хранит объекты (или любые другие значения, которые можно копировать, – смотрите врезку «Структурированные копии» выше). Каждый объект должен иметь *ключ*, под которым он сохраняется и извлекается из хранилища. Ключи должны быть уникальными – два объекта в одном хранилище не могут иметь одинаковые ключи, – и они должны иметь естественный порядок следования, чтобы их можно было сортировать. Примерами допустимых ключей являются строки, числа и объекты `Date`. База данных IndexedDB может автоматически генерировать уникальные ключи для каждого объекта, добавляемого в базу данных. Однако часто объекты, сохраняемые в хранилище объектов, уже будут иметь свойство, пригодное для использования в качестве ключа. В этом случае при создании хранилища объектов достаточно просто определить «путь к ключу», определяющий это свойство. Концептуально, путь к ключу – это значение, сообщающее базе данных, как извлечь ключ из объекта.

Помимо возможности извлекать объекты из хранилища по значению первичного ключа существует также возможность выполнить поиск по значениям других свойств объекта. Чтобы обеспечить эту возможность, в хранилище объектов можно определить любое количество *индексов*. (Способность индексировать объекты подчеркивается самим названием «IndexedDB».) Каждый индекс определяет вторичный ключ хранимых объектов. Эти индексы в целом могут быть неуникальными, и одному и тому же ключу может соответствовать множество объектов. Поэтому в операциях обращения к хранилищу объектов с использованием индекса обычно используется *курсор*, определяющий прикладной интерфейс для извлечения объектов из потока результатов по одному. Курсоры могут также использоваться для обращения к хранилищу объектов с использованием диапазона ключей (или индексов), и прикладной интерфейс IndexedDB включает объект, используемый для описания диапазонов (с верхней и/или с нижней границей, включающих или не включающих границы) ключей.

IndexedDB гарантирует атомарность операций: операции чтения и записи в базу данных объединяются в *транзакции*, благодаря чему либо они все будут успешно выполнены, либо ни одна из них не будет выполнена, и база данных никогда не останется в неопределенном, частично измененном состоянии. Транзакции в IndexedDB реализованы намного проще, чем во многих других прикладных интерфейсах к базам данных, и мы еще вернемся к ним ниже.

Концепция прикладного интерфейса IndexedDB чрезвычайно проста. Чтобы прочитать или изменить данные, сначала необходимо открыть требуемую базу данных (указав ее имя). Затем создать объект транзакции и с помощью этого объекта отыскать требуемое хранилище объектов в базе данных, также по имени. Наконец, отыскать объект вызовом метода `get()` хранилища объектов или сохранить новый объект вызовом метода `put()`. (Или вызвать метод `add()`, если необходимо избежать затирания существующих объектов.) Если потребуется отыскать объекты по диапазону ключей, нужно создать объект `IDBRange` и передать его методу `openCursor()` хранилища объектов. Или, если потребуется выполнить запрос по вторичному ключу, отыскать именованный индекс в хранилище объектов и затем вызвать метод `get()` или `openCursor()` объекта-индекса.

Однако эта концептуальная простота осложняется тем фактом, что прикладной интерфейс должен быть асинхронным, чтобы веб-приложения могли пользоваться им, не блокируя основной поток выполнения броузера, управляющий пользовательским интерфейсом. (Спецификация IndexedDB определяет синхронную версию прикладного интерфейса для использования в фоновых потоках выполнения, но на момент написания этих строк ни один броузер еще не реализовал эту версию, поэтому она не рассматривается здесь.) Создание транзакции, а также поиск хранилища объектов и индексов являются простыми синхронными операциями. Но открытие базы данных, обновление хранилища объектов с помощью метода `put()` и получение хранилища или индекса с помощью метода `get()` или `openCursor()` являются асинхронными операциями. Все эти асинхронные методы немедленно возвращают объект запроса. В случае успешного или неудачного выполнения запроса броузер генерирует событие «success» или «error» в объекте запроса, которые можно обработать, определив обработчики событий с помощью свойств `onsuccess` и `onerror`. В обработчике `onsuccess` результат операции доступен в виде свойства `result` объекта запроса.

Одно из удобств этого асинхронного прикладного интерфейса заключается в простоте управления транзакциями. При типичном использовании прикладного интерфейса IndexedDB сначала открывается база данных. Это асинхронная операция, поэтому по ее выполнении вызывается обработчик `onsuccess`. В этом обработчике создается объект транзакции, и затем этот объект используется для поиска хранилища или хранилищ объектов, которые предполагается использовать. После этого производится серия вызовов методов `get()` и `put()` хранилищ объектов. Они также действуют асинхронно, поэтому непосредственно при их вызове ничего не происходит, но запросы, сгенерированные этими методами `get()` и `put()`, автоматически будут связаны с объектом транзакции. При необходимости можно отменить все операции в транзакции, ожидающие выполнения, и откатить любые уже выполненные операции вызовом метода `abort()` объекта транзакции. Во многих других прикладных интерфейсах к базам данных объект транзакции обычно имеет метод `commit()`, подтверждающий транзакцию. Однако в IndexedDB транзакция подтверждается после выхода из обработчика `onsuccess`, создавшего

транзакцию, когда броузер вернется в цикл обработки событий, и после выполнения всех операций, запрошенных в транзакции (без запуска новых операций в их функциях обратного вызова). Такая схема, на первый взгляд, кажется слишком сложной, но в практическом применении она очень проста. При использовании прикладного интерфейса IndexedDB программист вынужден создавать объекты транзакций, чтобы получить доступ к хранилищам объектов, но в обычных ситуациях ему даже не приходится задумываться о транзакциях.

Наконец, существует один особый вид транзакций, обеспечивающий возможность работы очень важной части прикладного интерфейса IndexedDB. Создать новую базу данных с использованием интерфейса IndexedDB API очень просто: достаточно выбрать имя и запросить открытие этой базы данных. Но новая база данных создается абсолютно пустой, и она совершенно бесполезна, пока в нее не будет добавлено одно или более хранилищ объектов (и, возможно, нескольких индексов). Создавать хранилища объектов и индексы можно только внутри обработчика события `onsuccess` **объекта запроса, возвращаемого методом** `setVersion()` **объекта базы данных. Метод** `setVersion()` **позволяет указать номер версии базы данных – в обычной ситуации номер версии должен изменяться при каждом изменении структуры базы данных. Однако более важно, что метод** `setVersion()` **неявно запускает специальную транзакцию, позволяющую вызвать метод** `createObjectStore()` **объекта базы данных и метод** `createIndex()` **хранилища объектов.**

Теперь, получив представление о прикладном интерфейсе IndexedDB, вы сможете самостоятельно разобраться в примере 22.15. Этот пример использует IndexedDB для создания базы данных, отображающей почтовые индексы США в названия городов, и выполнения запросов к ней. Он демонстрирует многие, хотя и не все, основные особенности IndexedDB. На момент написания этих строк пример действовал в Firefox 4 и Chrome 11, но из-за того, что спецификация все еще продолжала меняться и реализации находились на предварительной стадии разработки, велика вероятность, что он не будет работать именно так, как описывается здесь, когда вы будете читать эти строки. Однако общая структура примера должна сохранить свою полезность для вас. Пример 22.15 получился достаточно длинным, но в нем имеется большое количество комментариев, которые облегчат его изучение.

Пример 22.15. База данных IndexedDB с почтовыми индексами США

```
<!DOCTYPE html>
<html>
<head>
<title>Zipcode Database</title>
<script>
// Реализации IndexedDB все еще используют префиксы в именах
var indexedDB = window.indexedDB ||     // Использовать стандартный API БД
    window.mozIndexedDB ||             // Или раннюю версию в Firefox
    window.webkitIndexedDB;            // Или раннюю версию в Chrome
// В Firefox не используются префиксы для следующих двух объектов:
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;

// Эта функция будет использоваться для вывода сообщений об ошибках
function logerr(e) {
    console.log("Ошибка IndexedDB " + e.code + ": " + e.message);
}
```

```
// Эта функция асинхронно получает объект базы данных (при необходимости
// создает и инициализирует базу данных) и передает его функции f().
function withDB(f) {
    var request = indexedDB.open("zipcodes"); // Открыть базу данных zipcode
    request.onerror = logerr;                 // Выводить сообщения об ошибках
    request.onsuccess = function() {          // Или вызвать эту функцию по завершении
        var db = request.result;  // Результатом запроса является база данных

        // Базу данных можно открыть всегда, даже если она не существует.
        // Мы проверяем версию, чтобы узнать, была ли БД создана и инициализирована.
        // Если нет - это необходимо сделать. Но если БД уже настроена,
        // остается просто передать ее функции f().
        if (db.version === "1") f(db); // Если БД существует, передать ее f()
        else initdb(db,f);             // Иначе сначала инициализировать ее
    }
}

// Принимает почтовый индекс, отыскивает город, которому он принадлежит,
// и асинхронно передает название города указанной функции.
function lookupCity(zip, callback) {
    withDB(function(db) {
        // Создать объект транзакции для этого запроса
        var transaction = db.transaction(["zipcodes"],  // Требуемое хранилище
                              IDBTransaction.READ_ONLY, // Не обновлять
                              0);                       // Время ожидания не ограничено
        // Получить хранилище объектов из транзакции
        var objects = transaction.objectStore("zipcodes");

        // Запросить объект, соответствующий указанному индексу.
        // Строки выше выполнялись синхронно, но эта выполняется асинхронно
        var request = objects.get(zip);
        request.onerror = logerr;         // Выводить сообщения об ошибках
        request.onsuccess = function() { // Передать результаты этой функции
            // Искомый объект сейчас в свойстве request.result
            var object = request.result;
            if (object) // Если соотв. найдено, передать город и штат функции
                callback(object.city + ", " + object.state);
            else        // Иначе сообщить о неудаче
                callback("Неизвестный индекс");
        }
    });
}

// По указанному названию города отыскивает все почтовые индексы для всех
// городов (в любом штате) с этим названием (с учетом регистра символов).
// Асинхронно передает результаты по одному указанной функции
function lookupZipcodes(city, callback) {
    withDB(function(db) {
        // Как и выше, создаем транзакцию и получаем хранилище объектов
        var transaction = db.transaction(["zipcodes"], IDBTransaction.READ_ONLY, 0);
        var store = transaction.objectStore("zipcodes");
        // На этот раз нам требуется получить индекс по названиям городов
        var index = store.index("cities");

        // Этот запрос может вернуть несколько результатов, поэтому, чтобы
        // получить их все, следует использовать объект курсора. Чтобы создать
        // курсор, нужно создать объект диапазона, представляющий диапазон ключей
        var range = new IDBKeyRange.only(city); // Диапазон с одним ключом
        // Все, что выше, выполняется синхронно.
        // Теперь нужно запросить курсор, который возвращается асинхронно.
        var request = index.openCursor(range); // Запросить курсор
        request.onerror = logerr;              // Сообщать об ошибках
        request.onsuccess = function() { // Передать курсор этой функции
            // Этот обработчик событий будет вызван несколько раз, по одному
            // для каждой записи, соответствующей запросу, и еще один раз
            // с пустым курсором, указывающим на окончание результатов.
            var cursor = request.result // Курсор в свойстве request.result
            if (!cursor) return;        // Нет курсора = нет результатов
            var object = cursor.value   // Получить совпавшую запись
            callback(object);           // Передать ее указанной функции
            cursor.continue();          // Запросить следующую запись
        };
    });
}

// Эта функция используется обработчиком onchange в документе ниже.
// Она выполняет запрос к БД и отображает результаты
function displayCity(zip) {
    lookupCity(zip, function(s) { document.getElementById('city').value=s; });
}

// Это другая функция, используемая обработчиком onchange в документе ниже.
// Она выполняет запрос к БД и отображает результаты
function displayZipcodes(city) {
    var output = document.getElementById("zipcodes");
    output.innerHTML = "Найденные индексы:";
    lookupZipcodes(city, function(o) {
        var div = document.createElement("div");
        var text = o.zipcode + ": " + o.city + ", " + o.state;
        div.appendChild(document.createTextNode(text));
        output.appendChild(div);
    });
}

// Настраивает структуру базы данных и заполняет ее данными, затем передает базу данных
// функции f(). Эта функция вызывается функцией withDB(), если база данных еще не была
// инициализирована. Это самая хитрая часть программы, поэтому мы оставили ее напоследок.
function initdb(db, f) {
    // Загрузка информации о почтовых индексах и сохранение ее в базе данных может
    // потребовать некоторого дополнительного времени при первом запуске этого
    // приложения. Поэтому необходимо вывести сообщение, извещающее о выполнении операции.
    var statusline = document.createElement("div");
    statusline.style.cssText =
        "position:fixed; left:0px; top:0px; width:100%;" +
        "color:white; background-color: black; font: bold 18pt sans-serif;" +
        "padding: 10px; ";
    document.body.appendChild(statusline);
    function status(msg) { statusline.innerHTML = msg.toString(); };

    status("Инициализация базы данных почтовых индексов");

    // Единственное место, где можно определить или изменить структуру
    // базы данных IndexedDB - обработчик onsucess запроса setVersion.
    var request = db.setVersion("1"); // Попробовать изменить версию БД
    request.onerror = status;         // Вывести сообщение в случае ошибки
    request.onsuccess = function() {  // Иначе вызвать эту функцию
```

```
// База данных почтовых индексов включает единственное хранилище.
// Оно хранит объекты следующего вида: {
//    zipcode: "02134", // Отправьте на телепередачу Zoom!¹ :-)
//    city: "Allston",
//    state: "MA",
//    latitude: "42.355147",
//    longitude: "-71.13164"
// }
//
// Свойство "zipcode" используется в качестве ключа базы данных
// Кроме того, создается индекс по названию города

// Создать хранилище объектов, указав имя хранилища и объект с параметрами,
// включающий "путь к ключу", определяющий имя свойства-ключа для этого
// хранилища. (Если опустить путь к ключу, IndexedDB определит свой
// собственный уникальный целочисленный ключ.)
var store = db.createObjectStore("zipcodes", // имя хранилища
                                 { keyPath: "zipcode" });

// Создать в хранилище объектов индекс по названию города.
// Строка пути к ключу передается этому методу непосредственно,
// как обязательный аргумент, а не как свойство объекта с параметрами.
store.createIndex("cities", "city");

// Теперь необходимо загрузить информацию о почтовых индексах, преобразовать
// ее в объекты и сохранить эти объекты в созданном выше хранилище.
//
// Файл с исходными данными содержит строки следующего вида:
//
// 02130,Jamaica Plain,MA,42.309998,-71.11171
// 02131,Roslindale,MA,42.284678,-71.13052
// 02132,West Roxbury,MA,42.279432,-71.1598
// 02133,Boston,MA,42.338947,-70.919635
// 02134,Allston,MA,42.355147,-71.13164
//
// Как ни странно, но почтовая служба США не обеспечивает свободный доступ
// к этой информации, поэтому мы будет использовать устаревшие данные переписи
// с сайта: http://mappinghacks.com/2008/04/28/civicspace-zip-code-database/
// Для загрузки данных используется объект XMLHttpRequest.
// Но для обработки данных по мере поступления будут использованы
// новые события onload и onprogress, определяемые спецификацией XHR2
var xhr = new XMLHttpRequest();  // Объект XHR для загрузки данных
xhr.open("GET", "zipcodes.csv"); // HTTP-запрос типа GET для этого URL
xhr.send();                      // Запустить немедленно
xhr.onerror = status;            // Отображать сообщения об ошибках
```

---

[1] В 1972–1978 гг. телекомпания WGBH-TV в Бостоне выпускала детское телешоу «ZOOM». Детям предлагалось после шоу «выключить телевизор и сделать то, о чем рассказывалось в передаче...». Дети, обычно семеро, участвовавшие в шоу, играли в игры, ставили пьесы, рассказывали стихи, ставили научные эксперименты, вели беседы на такие темы, как больницы, предрассудки и другие, предлагаемые телезрителями. Передача имела призыв со словами: «Напишите на конверте ZOOM, Зет-Дабл-Оу-М, Бокс 3-5-0, Бостон, Масс 0-2-1-3-4 и отправьте его на ZOOM!». Весь текст проговаривался, кроме индекса, который пропевался. Описание в Википедии: *http://en.wikipedia.org/wiki/ZOOM_(1972_TV_series)*. – *Прим. перев.*

```
var lastChar = 0, numlines = 0;  // Уже обработанный объем

// Обрабатывает файл базы данных блоками, по мере загрузки
xhr.onprogress = xhr.onload = function(e) { // Сразу два обработчика!
    // Обработать блок между lastChar и последним принятым символом
    // перевода строки. (Нам требуется отыскать последний символ
    // перевода строки, чтобы не обработать неполную запись)
    var lastNewline = xhr.responseText.lastIndexOf("\n");
    if (lastNewline > lastChar) {
        var chunk = xhr.responseText.substring(lastChar, lastNewline)
        lastChar = lastNewline + 1; // Откуда начинать в следующий раз

        // Разбить новый фрагмент на строки
        var lines = chunk.split("\n");
        numlines += lines.length;

        // Чтобы вставить информацию о почтовом индексе в базу данных, необходимо
        // получить объект транзакции. Все операции добавления объектов
        // в базу данных, выполняемые с использованием этого объекта,
        // будут автоматически подтверждаться после выхода из этой функции,
        // когда броузер вернется в цикл обработки событий.
        // Чтобы создать объект транзакции, следует определить,
        // какие хранилища объектов будут использоваться (у нас имеется всего
        // одно хранилище). Кроме того, требуется сообщить, что будет
        // выполняться не только чтение, но и запись в базу данных:
        var transaction = db.transaction(["zipcodes"], // хранилища
                                         IDBTransaction.READ_WRITE);

        // Получить ссылку на хранилище из объекта транзакции
        var store = transaction.objectStore("zipcodes");

        // Теперь обойти в цикле строки в файле с почтовыми индексами,
        // создать на их основе объекты и добавить их в хранилище.
        for(var i = 0; i < lines.length; i++) {
            var fields = lines[i].split(",");// Значения через запятую
            var record = {                   // Сохраняемый объект
                zipcode: fields[0],          // Все свойства - строки
                city: fields[1],
                state: fields[2],
                latitude: fields[3],
                longitude: fields[4]
            };

            // Вся прелесть IndexedDB API в том, что хранилище
            // объектов *по-настоящему* просто использовать.
            // Следующая строка добавляет запись:
            store.put(record); // Или add(), чтобы избежать затирания
        }

        status("Инициализация базы данных, загружено записей: "
              + numlines + ".");
    }

    if (e.type == "load") {
        // Если это было последнее событие load, значит, мы отправили в базу
        // данных все сведения о почтовых индексах. Но, так как мы только
        // что обработали порядка 40000 записей, они все еще могут записываться
        // в хранилище. Поэтому мы выполним простой запрос. Когда он будет
```

```
                    // успешно выполнен, это послужит сигналом, что база данных готова
                    // к работе, и наконец можно удалить строку сообщения и вызвать
                    // функцию f(), которая так давно была передана функции withDB()
                    lookupCity("02134", function(s) { // Allston, MA
                        document.body.removeChild(statusline);
                        withDB(f);
                    });
                }
            }
        }
    }
</script>
</head>
<body>
<p>Введите почтовый индекс, чтобы отыскать город:</p>
Индекс: <input onchange="displayCity(this.value)"></input>
Город: <output id="city"></output>
</div>
<div>
<p>Введите название города (с учетом регистра символов, без названия штата),
чтобы отыскать все города с этим названием и их почтовые индексы:</p>
Город: <input onchange="displayZipcodes(this.value)"></input>
<div id="zipcodes"></div>
</div>
<p><i>Этот пример работает только в Firefox 4 и Chrome 11.</i></p>
<p><i>Выполнение первого запроса может занять длительное время.</i></p>
<p><i>Вам может потребоваться запустить Chrome с ключом --unlimited-quota-for-
indexeddb</i></p>
</body>
</html>
```

22.9. Веб-сокеты

В главе 18 демонстрируется, как клиентские сценарии на языке JavaScript могут взаимодействовать с серверами по сети. Все примеры в этой главе используют протокол HTTP, а это означает, что все они ограничены исходной природой протокола HTTP: этот протокол, не имеющий информации о состоянии, состоит из запросов клиента и ответов сервера. Протокол HTTP фактически является узкоспециализированным сетевым протоколом. Более универсальные сетевые взаимодействия через Интернет (или через локальные сети) часто реализуются с использованием долгоживущих соединений и обеспечивают двунаправленный обмен сообщениями через TCP-сокеты. Довольно небезопасно предоставлять клиентскому сценарию на языке JavaScript доступ к низкоуровневым TCP-сокетам, однако спецификация «WebSocket API» определяет безопасную альтернативу: она позволяет клиентским сценариям создавать двунаправленные соединения с серверами, поддерживающими протокол веб-сокетов. Это существенно упрощает решение некоторых сетевых задач.

Прикладной интерфейс веб-сокетов удивительно прост в использовании. Сначала необходимо создать сокет с помощью конструктора `WebSocket()`:

```
var socket = new WebSocket("ws://ws.example.com:1234/resource");
```

Протокол веб-сокетов

Чтобы использовать веб-сокеты в сценариях на языке JavaScript, достаточно будет освоить клиентский прикладной интерфейс веб-сокетов, описываемый здесь. Не существует эквивалентного серверного прикладного интерфейса для создания серверов с поддержкой веб-сокетов; в этом разделе будет представлен простой пример сервера, основанного на использовании интерпретатора Node (раздел 12.2) и сторонней серверной библиотеки поддержки веб-сокетов. Клиент и сервер взаимодействуют через долгоживущие TCP-сокеты, следуя правилам, определяемым протоколом веб-сокетов. Мы не будем рассматривать здесь особенности протокола, но следует отметить, что протокол веб-сокетов спроектирован очень аккуратно, благодаря чему веб-серверы могут легко обрабатывать HTTP-соединения и соединения на основе веб-сокетов через один и тот же порт.

Веб-сокеты получили широкую поддержку среди производителей броузеров. В ранней, предварительной версии протокола веб-сокетов была обнаружена серьезная брешь в системе безопасности, поэтому на момент написания этих строк в некоторых броузерах поддержка веб-сокетов была отключена, – до стандартизации безопасной версии протокола. В Firefox 4, например, может потребоваться явно включить поддержку веб-сокетов, открыв страницу `about:config` и установив переменную «network.websocket.override-security-block» в значение `true`.

Аргументом конструктора `WebSocket()` является URL-адрес, в котором используется протокол `ws://` (или `wss://` – в случае с безопасными соединениями, по аналогии с `https://`). URL-адрес определяет имя хоста, к которому выполняется подключение, и может также определять порт (по умолчанию веб-сокеты используют тот же порт, что и протоколы HTTP и HTTPS) и путь или ресурс.

После создания сокета в нем обычно регистрируются обработчики событий:

```
socket.onopen = function(e) {  /* Соединение установлено. */ };
socket.onclose = function(e) { /* Соединение закрыто. */ };
socket.onerror = function(e) { /* Что-то пошло не так! */ };
socket.onmessage = function(e) {
var message = e.data;          /* Сервер послал сообщение. */
};
```

Чтобы отправить данные серверу через сокет, следует вызвать метод `send()` сокета:

```
socket.send("Привет, сервер!");
```

Текущая версия прикладного интерфейса веб-сокетов поддерживает только текстовые сообщения и отправляет их в виде строк в кодировке UTF-8. Однако текущая версия спецификации протокола веб-сокетов включает поддержку двоичных сообщений, и будущие версии прикладного интерфейса, возможно, будут обеспечивать обмен двоичными данными с сервером.

По окончании взаимодействия с сервером сценарий может закрыть веб-сокет вызовом его метода `close()`.

Веб-сокеты являются двунаправленными, и единственное соединение, установленное через веб-сокет, может использоваться клиентом и сервером для передачи сообщений друг другу в любой момент времени. Это взаимодействие не обязательно должно иметь форму запросов и ответов. Каждая служба, основанная на веб-сокетах, будет определять собственный «подпротокол» передачи данных между клиентом и сервером. С течением времени эти «подпротоколы» могут развиться, и вам может потребоваться реализовать клиенты и серверы, поддерживающие несколько версий подпротокола. К счастью, протокол веб-сокетов включает механизм, дающий возможность договориться о выборе подпротокола, который поддерживается и клиентом, и сервером. Конструктору `WebSocket()` можно передать массив строк. Сервер получит его в виде списка подпротоколов, поддерживаемых клиентом. Сервер выберет подпротокол, поддерживаемый им, и отправит его обратно клиенту. После установления соединения клиент сможет определить, какой подпротокол можно использовать, проверив свойство `protocol` объекта WebSocket.

В разделе 18.3 описывается прикладной интерфейс объекта `EventSource` и демонстрируется его применение на примере реализации клиента и сервера чата. Веб-сокеты еще больше упрощают реализацию подобных приложений. В примере 22.16 демонстрируется очень простой клиент чата: он напоминает пример 18.15, но использует веб-сокеты для двунаправленного обмена сообщениями вместо объекта `EventSource` для приема сообщений и `XMLHttpRequest` – для отправки.

Пример 22.16. Клиент чата на основе веб-сокетов

```
<script>
window.onload = function() {
    // Позаботиться о некоторых деталях пользовательского интерфейса
    var nick = prompt("Введите свой псевдоним");  // Получить псевдоним
    var input = document.getElementById("input"); // Отыскать поле ввода
    input.focus();                                // Установить фокус ввода

    // Открыть веб-сокет для отправки и приема сообщений в чате.
    // Предполагается, что HTTP-сервер, откуда загружается сценарий, также
    // поддерживает веб-сокеты, и для связи с ним используется то же имя хоста
    // и номер порта, но вместо протокола http:// используется протокол ws://
    var socket = new WebSocket("ws://" + location.host + "/");

    // Так через веб-сокет принимаются сообщения с сервера
    socket.onmessage = function(event) { // Вызывается при получении сообщения
        var msg = event.data;           // Получить текст из объекта события
        var node = document.createTextNode(msg); // Создать текстовый узел
        var div = document.createElement("div"); // Создать элемент <div>
        div.appendChild(node);                   // Добавить текстовый узел
        document.body.insertBefore(div, input);  // и вставить div перед полем ввода
        input.scrollIntoView(); // Гарантировать видимость элемента input
    }

    // Так через веб-сокет отправляются сообщения на сервер
    input.onchange = function() { // Когда пользователь нажмет клавишу Enter
        var msg = nick + ": " + input.value; // Имя пользователя и текст
        socket.send(msg);                    // Отправить через сокет
        input.value = ""; // Подготовиться к вводу следующего сообщения
    }
};
</script>
<!-- Пользовательский интерфейс - это единственное поле ввода -->
```

```
<!-- Новые сообщения в чате будут появляться перед этим элементом -->
<input id="input" style="width:100%"/>
```

В примере 22.17 демонстрируется реализация сервера чата, основанного на веб-сокетах, которая предназначена для работы под управлением интерпретатора Node (раздел 12.2). Сравните этот пример с примером 18.17, чтобы увидеть, что веб-сокеты упрощают не только клиентскую часть реализации чата, но и серверную.

Пример 22.17. Сервер чата на основе веб-сокетов, выполняющийся под управлением Node

```
/*
 * Этот серверный сценарий на языке JavaScript предназначен для выполнения
 * под управлением NodeJS. Он играет роль сервера веб-сокетов, реализованного поверх
 * HTTP-сервера с использованием внешней библиотеки websocket, которую можно найти
 * по адресу: https://github.com/miksago/node-websocket-server/. При обращении
 * к ресурсу "/" он возвращает HTML-файл клиента чата. В ответ на обращение к любому
 * другому ресурсу возвращается код 404. Сообщения принимаются посредством протокола
 * веб-сокетов и просто рассылаются по всем активным соединениям.
 */
var http = require('http');           // Использовать HTTP-сервер в Node
var ws = require('websocket-server'); // Использовать внешнюю библиотеку

// Прочитать исходный файл с реализацией клиента чата. Используется ниже.
var clientui = require('fs').readFileSync("wschatclient.html");

// Создать HTTP-сервер
var httpserver = new http.Server();

// Когда HTTP-сервер получит новый запрос, будет вызвана эта функция
httpserver.on("request", function (request, response) {
    // Если запрошен ресурс "/", отправить реализацию клиента чата.
    if (request.url === "/") { // Запрошена реализация клиента чата
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write(clientui);
        response.end();
    }
    else { // В ответ на любой другой запрос отправить код 404 "Not Found"
        response.writeHead(404);
        response.end();
    }
});

// Обернуть HTTP-сервер сервером на основе веб-сокетов
var wsserver = ws.createServer({server: httpserver});

// Вызывать эту функцию при получении новых запросов на соединение
wsserver.on("connection", function(socket) {
    socket.send("Добро пожаловаь в чат."); // Приветствовать нового клиента
    socket.on("message", function(msg) {   // Принимать сообщения от клиента
        wsserver.broadcast(msg);           // И рассылать их всем остальным
    });
});

// Запустить сервер на порту 8000. Запуск сервера на основе веб-сокетов
// приводит также к запуску HTTP-сервера. Для его использования подключайтесь
// по адресу http://localhost:8000/.
wsserver.listen(8000);
```

III

Справочник по базовому JavaScript

Эта часть книги представляет собой справочник по всем классам, свойствам и методам базового прикладного программного интерфейса JavaScript. В справочнике описываются следующие классы и объекты, в алфавитном порядке:

- `Arguments`
- `Array`
- `Boolean`
- `Date`
- `Error`
- `EvalError`
- `Function`
- `Global`
- `JSON`
- `Math`
- `Number`
- `Object`
- `RangeError`
- `ReferenceError`
- `RegExp`
- `String`
- `SyntaxError`
- `TypeError`
- `URIError`

Справочные страницы с описанием методов и свойств классов отсортированы по их полным именам, включающим имена определяющих их классов. Например, чтобы найти метод `replace()` класса `String`, его следует искать как `String.replace()`, а не просто `replace`.

В базовом JavaScript определены некоторые глобальные функции и свойства, такие как `eval()` и `NaN`. Формально они являются свойствами глобального объекта. Однако у глобального объекта нет имени, поэтому в справочнике они перечислены по их неполным именам. Для удобства полный набор глобальных функций и свойств базового JavaScript объединен в специальную справочную статью «Global» (хотя объекта или класса с таким именем нет).

Справочник по базовому JavaScript

arguments[]

массив аргументов функции

Синтаксис

`arguments`

Описание

Массив `arguments[]` определен только внутри тела функции, где он ссылается на объект `Arguments` этой функции. Данный объект имеет нумерованные свойства и представляет собой массив, содержащий все переданные функции аргументы. Идентификатор `arguments` – это, по существу, локальная переменная, автоматически объявляемая и инициализируемая внутри каждой функции. Она ссылается на объект `Arguments` только внутри тела функции и не определена в глобальном программном коде.

См. также

`Arguments`; глава 8

Arguments

аргументы и другие свойства функции **Object→Arguments**

Синтаксис

`arguments`
`arguments[`*`n`*`]`

Элементы

Объект `Arguments` определен только внутри тела функции. Хотя формально он не является массивом, у него есть нумерованные свойства, действующие как элементы массива, и свойство `length`, значение которого равно количеству элементов массива. Его элементами являются значения, переданные функции в качестве аргументов. Элемент 0 – это первый аргумент, элемент 1 – второй аргумент и т. д. Все значения, переданные в качестве аргументов, становятся элементами массива в объекте `Arguments` независимо от того, присвоены ли этим аргументам имена в объявлении функции.

Свойства

`callee`	Ссылка на выполняемую в данный момент функцию.
`length`	Количество аргументов, переданных функции, и количество элементов массива в объекте `Arguments`.

Описание

Когда вызывается функция, для нее создается объект `Arguments`, и локальная переменная `arguments` автоматически инициализируется ссылкой на объект `Arguments`. Основное назначение объекта `Arguments` – предоставить возможность определить, сколько аргументов передано функции, и обратиться к неименованным аргументам. В дополнение к элементам массива и свойству `length`, у объекта `Arguments` имеется свойство `callee`, которое позволяет неименованной функции ссылаться на саму себя.

Для большинства задач объект `Arguments` можно рассматривать как массив с дополнительным свойством `callee`. Однако он не является экземпляром объекта `Array`, а свойство `Arguments.length` не обладает особым поведением, как свойство `Array.length`, и не может использоваться для изменения размера массива.

При выполнении сценария в нестрогом режиме объект `Arguments` имеет одну *очень* необычную особенность. Когда у функции есть именованные аргументы, элементы массива в объекте `Arguments` являются синонимами локальных переменных, содержащих аргументы функции. Объект `Arguments` и имена аргументов предоставляют два различных способа обращения к одной и той же переменной. Изменение значения аргумента с помощью имени аргумента изменяет значение, извлекаемое через объект `Arguments`, а изменение значения аргумента через объект `Arguments` изменяет значение, извлекаемое по имени аргумента.

См. также

`Function`; глава 8

Arguments.callee

не доступно в строгом режиме

функция, выполняемая в данный момент

Синтаксис

`arguments.callee`

Описание

`arguments.callee` ссылается на функцию, выполняющуюся в данный момент. Данный синтаксис предоставляет неименованной функции возможность сослаться на себя. Это свойство определено только внутри тела функции.

Пример

```
// Неименованный литерал функции использует свойство callee
// для ссылки на себя, чтобы произвести рекурсивный вызов
var factorial = function(x) {
    if (x < 2) return 1;
    else return x * arguments.callee(x1);
}
var y = factorial(5); // Вернет 120
```

Arguments.length

число аргументов, переданных функции

Синтаксис

`arguments.length`

Описание

Свойство `length` объекта `Arguments` возвращает количество аргументов, переданных текущей функции. Это свойство определено только внутри тела функции.

Обратите внимание: это свойство возвращает фактическое количество переданных аргументов, а не ожидаемое. О количестве аргументов в объявлении функции говорится в справочной статье `Function.length`. Кроме того, следует отметить, что это свойство не обладает особым поведением, как свойство `Array.length`.

Пример

```
// Использовать объект Arguments, чтобы проверить, верное ли количество
// аргументов было передано
function check(args) {
    var actual = args.length;          // Фактическое количество аргументов
    var expected = args.callee.length; // Ожидаемое количество аргументов
    if (actual != expected) {          // Если не совпадают, сгенерировать исключение
        throw new Error("Неверное число аргументов: ожидается: " +
                        expected + "; фактически передано " + actual);
    }
}
// Функция, демонстрирующая использование функции, приведенной выше
function f(x, y, z) {
    check(arguments); // Проверить правильность количества аргументов
    return x + y + z; // Выполнить оставшуюся часть функции обычным образом
}
```

См. также

`Array.length`, `Function.length`

Array

встроенная поддержка массивов **Object→Array**

Конструктор

`new Array()`

`new Array(`*размер*`)`

`new Array(`*элемент0*, *элемент1*, ..., *элементn*`)`

Аргументы

размер

Желаемое количество элементов в массиве. Длина возвращаемого массива (`length`) равна аргументу *размер*.

элемент0, ... элементn

Список аргументов из двух и более произвольных значений. Когда конструктор `Array()` вызывается с этими аргументами, элементы только что созданного массива инициализируются указанными значениями, а свойство `length` становится равным количеству аргументов.

Возвращаемое значение

Вновь созданный и инициализированный массив. Когда конструктор `Array()` вызывается без аргументов, он возвращает пустой массив, свойство `length` которого равно 0.

При вызове с одним числовым аргументом конструктор возвращает массив с указанным количеством неопределенных элементов. При вызове с любыми другими аргументами конструктор инициализирует массив значениями аргументов. Когда конструктор `Array()` вызывается как функция (без оператора `new`), он ведет себя точно так же, как при вызове с оператором `new`.

Исключения

`RangeError`	Когда конструктору `Array()` передается один целый аргумент *размер*, генерируется исключение `RangeError`, если размер отрицателен или превышает 2^{32}-1.

Синтаксис литерала

ECMAScript v3 определяет синтаксис литералов для массивов. Программист может создавать и инициализировать массив, заключая список выражений, перечисленных через запятые, в квадратные скобки. Значения этих выражений становятся элементами массива. Например:

```
var a = [1, true, 'abc'];
var b = [a[0], a[0]*2, f(x)];
```

Свойства

`length`	Целое, доступное для чтения и записи, определяет количество элементов массива или, если элементы массива расположены не непрерывно, число, на единицу большее индекса последнего элемента массива. Изменение этого свойства укорачивает или расширяет массив.

Методы

Методы `every()`, `filter()`, `forEach()`, `indexOf()`, `lastIndexOf()`, `map()`, `reduce()`, `reduceRight()` и `some()` впервые появились в ECMAScript 5, но были реализованы всеми броузерами, кроме IE, до утверждения стандарта ES5.

`concat()`	Присоединяет элементы к массиву.
`every()`	Проверяет, возвращает ли предикат значение `true` для каждого элемента массива.
`filter()`	Возвращает массив элементов, удовлетворяющих требованиям функции-предиката.
`forEach()`	Вызывает функцию для каждого элемента массива.
`indexOf()`	Выполняет поиск элемента в массиве.
`join()`	Преобразует все элементы массива в строки и выполняет их конкатенацию.
`lastIndexOf()`	Выполняет поиск в массиве в обратном порядке.
`map()`	Вычисляет элементы нового массива из элементов данного массива.
`pop()`	Удаляет элемент из конца массива.
`push()`	Помещает элемент в конец массива.
`reduce()`	Вычисляет значение на основе элементов данного массива.
`reduceRight()`	Выполняет свертку массива справа налево.
`reverse()`	Меняет порядок следования элементов в массиве на противоположный.
`shift()`	Сдвигает элементы к началу массива.
`slice()`	Возвращает подмассив массива.

`some()`	Проверяет, возвращает ли предикат значение `true` хотя бы для одного элемента массива.
`sort()`	Сортирует элементы массива.
`splice()`	Вставляет, удаляет или заменяет элементы массива.
`toLocaleString()`	Преобразует массив в локализованную строку.
`toString()`	Преобразует массив в строку.
`unshift()`	Вставляет элементы в начало массива.

Описание

Массивы – это базовое средство JavaScript, подробно описанное в главе 7.

См. также

Глава 7

Array.concat()

выполняет конкатенацию массивов

Синтаксис

массив.`concat`(*значение*, ...)

Аргументы

значение, ...	Любое количество значений, присоединяемых к *массиву*.

Возвращаемое значение

Новый массив, образуемый присоединением к *массиву* каждого из указанных аргументов.

Описание

Метод `concat()` создает и возвращает новый массив, являющийся результатом присоединения каждого из его аргументов к *массиву*. Этот метод не изменяет *массив*. Если какие-либо из аргументов `concat()` сами являются массивами, то присоединяются элементы этих массивов, а не сами массивы.

Пример

```
var a = [1,2,3];
a.concat(4, 5)          // Вернет [1,2,3,4,5]
a.concat([4,5]);        // Вернет [1,2,3,4,5]
a.concat([4,5],[6,7])   // Вернет [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]])  // Вернет [1,2,3,4,5,[6,7]]
```

См. также

`Array.join()`, `Array.push()`, `Array.splice()`

Array.every() EcmaScript 5

проверяет, возвращает ли предикат значение true для каждого элемента массива

Синтаксис

массив.`every`(*предикат*)

массив.`every`(*предикат*, *о*)

Аргументы

предикат	Функция-предикат, выполняющая проверку элементов *массива*
o	Необязательное значение `this`, передаваемое *предикату*.

Возвращаемое значение

`true`, если *предикат* вернет `true` (или какое-либо истинное значение) для каждого элемента *массива*, или `false`, если *предикат* вернет `false` (или какое-либо ложное значение) хотя бы для одного элемента *массива*.

Описание

Метод `every()` проверяет соответствие всех элементов массива некоторому условию. Он обходит в цикле элементы *массива* в порядке возрастания индексов и для каждого элемента вызывает указанную функцию *предикат*. Если *предикат* вернет `false` (или любое другое значение, которое в логическом контексте преобразуется в значение `false`), метод `every()` прекратит выполнение цикла и немедленно вернет `false`. Если для каждого элемента *предикат* вернет `true`, то и метод `every()` вернет `true`. При применении к пустому массиву `every()` возвращает `true`.

Для каждого индекса `i` в массиве функция *предикат* вызывается с тремя аргументами:

```
предикат(массив[i], i, массив)
```

Возвращаемое значение функции *предиката* интерпретируется как логическое значение. Значение `true` и все истинные значения указывают, что элемент массива прошел проверку или соответствует условию, описываемому этой функцией. Значение `false` или любое ложное значение означает, что элемент массива не прошел проверку.

Дополнительные сведения приводятся в статье `Array.forEach()`.

Пример

```
[1,2,3].every(function(x) { return x < 5; }) // => true: все элементы < 5
[1,2,3].every(function(x) { return x < 3; }) // => false: не все элементы < 3
[].every(function(x) { return false; });     // => true: всегда true для []
```

См. также

`Array.filter()`, `Array.forEach()`, `Array.some()`

Array.filter() ECMAScript 5

возвращает элементы массива, пропущенные предикатом

Синтаксис

массив.`map`(*предикат*)

массив.`map`(*предикат*, *o*)

Аргументы

предикат	Функция, которая определяет, может ли данный элемент *массива* быть включен в возвращаемый массив.
o	Необязательное значение `this`, передаваемое *предикату*.

Возвращаемое значение

Новый массив, содержащий только те элементы *массива*, для которых *предикат* вернет `true` (или истинное значение).

Описание

Метод `filter()` создает новый массив и заполняет его элементами *массива*, для которых функция *предикат* вернет `true` (или истинное значение). Метод `filter()` не изменяет сам *массив* (хотя функция *предикат* может делать это).

Метод `filter()` выполняет цикл по индексам *массива* в порядке возрастания и вызывает *предикат* для каждого элемента. Для каждого индекса `i` функция *предикат* вызывается с тремя аргументами:

```
предикат(массив[i], i, массив)
```

Если *предикат* вернет `true` или истинное значение, элемент с индексом `i` в *массиве* будет добавлен во вновь созданный массив. После того как метод `filter()` проверит все элементы *массива*, он вернет новый массив.

Дополнительные сведения приводятся в статье `Array.forEach()`.

Пример

```
[1,2,3].filter(function(x) { return x > 1; }); // => [2,3]
```

См. также

`Array.every()`, `Array.forEach()`, `Array.indexOf()`, `Array.map()`, `Array.reduce()`

Array.forEach() ECMAScript 5

вызывает функцию для каждого элемента массива

Синтаксис

массив.`forEach(`*f*`)`
массив.`forEach(`*f*, *o*`)`

Аргументы

f	Функция, вызываемая для каждого элемента *массива*.
o	Необязательное значение `this`, передаваемое функции *f*.

Возвращаемое значение

Этот метод ничего не возвращает.

Описание

Метод `forEach()` выполняет цикл по индексам *массива* в порядке возрастания и вызывает функцию *f* для каждого элемента. Для каждого индекса `i` функция *f* вызывается с тремя аргументами:

```
f(массив[i], i, массив)
```

Значение, возвращаемое функцией *f*, игнорируется. Обратите внимание, что метод `forEach()` ничего не возвращает. В частности, он не возвращает массив.

Особенности методов массивов

Описываемые ниже особенности относятся к методу `forEach()`, а также к родственным ему методам `map()`, `filter()`, `every()` и `some()`.

Каждый из этих методов принимает функцию в первом аргументе и необязательный второй аргумент. Если указан второй аргумент *o*, функция будет вызвана как метод объекта *o*. То есть в теле функции ключевое слово `this` будет возвращать *o*. Если второй аргумент не указан, то функция будет вызываться как функция (а не как метод) и ключевое слово `this` в ней будет ссылаться на глобальный объект при выполнении в нестрогом режиме или содержать значение `null` при выполнении в строгом режиме.

Каждый из этих методов проверяет длину *массива* перед началом итераций. Если вызываемая функция добавляет новые элементы в конец *массива*, цикл по этим новым элементам выполняться не будет. Если функция изменяет существующие элементы, цикл по которым еще не выполнялся, на следующих итерациях она получит измененные значения.

При работе с разреженными массивами эти методы не вызывают функцию для индексов с фактически отсутствующими элементами.

Пример

```
var a = [1,2,3];
a.forEach(function(x,i,a) { a[i]++; }); // а теперь будет [2,3,4]
```

См. также

`Array.every()`, `Array.filter()`, `Array.indexOf()`, `Array.map()`, `Array.reduce()`

Array.indexOf() ECMAScript 5

поиск в массиве

Синтаксис

массив.`indexOf(`*значение*`)`

массив.`indexOf(`*значение*, *начало*`)`

Аргументы

значение	Значение, которое ищется в *массиве*.
начало	Необязательный индекс элемента, с которого следует начать поиск. Если отсутствует, по умолчанию поиск начинается с индекса 0.

Возвращаемое значение

Первый индекс >= *началу* в *массиве*, где элемент === *значению*, или -1, если такой элемент не найден.

Описание

Этот метод выполняет поиск в *массиве* элемента, эквивалентного указанному *значению*, и возвращает индекс первого найденного элемента. Поиск начинается с индекса, определяемого аргументом *начало*, или с 0 и продолжается в порядке последовательного увеличения индексов, пока не будет найдено соответствие или пока не будут проверены все элементы. Для проверки эквивалентности используется оператор ===. Возвращает индекс первого соответствующего элемента или -1, если соответствие не было найдено.

Пример

```
['a','b','c'].indexOf('b')   // => 1
['a','b','c'].indexOf('d')   // => -1
['a','b','c'].indexOf('a',1) // => -1
```

См. также

`Array.lastIndexOf()`, `String.indexOf()`

Array.join()

выполняет конкатенацию элементов массива в строку

Синтаксис

массив.`join()`

массив.`join(`*разделитель*`)`

Аргументы

разделитель	Необязательный символ или строка, выступающая в качестве разделителя элементов в результирующей строке. Если аргумент опущен, используется запятая.

Возвращаемое значение

Строка, получающаяся в результате преобразования каждого элемента массива в строку и объединения их с разделителем между элементами путем конкатенации.

Описание

Метод `join()` преобразует каждый элемент *массива* в строку и затем выполняет конкатенацию этих строк, вставляя указанный *разделитель* между элементами. Возвращает полученную строку.

Обратное преобразование (разбиение строки на элементы массива) можно выполнить с помощью метода `split()` объекта `String`. Подробности см. в справочной статье `String.split()`.

Пример

```
a = new Array(1, 2, 3, "testing");
s = a.join("+"); // s - это строка "1+2+3+testing"
```

См. также

`String.split()`

Array.lastIndexOf() ECMAScript 5

выполняет поиск в массиве в обратном порядке

Синтаксис

массив.`lastIndexOf(`*значение*`)`

массив.`lastIndexOf(`*значение*, *начало*`)`

Аргументы

значение	Искомое значение.
начало	Необязательный индекс элемента, с которого следует начать поиск. Если отсутствует, по умолчанию поиск начинается с последнего элемента *массива*.

Возвращаемое значение

Наибольший индекс <= *начало* в *массиве*, где элемент === *значение*, или -1, если такой элемент не найден.

Описание

Этот метод выполняет поиск элемента, эквивалентного указанному *значению*, в обратном порядке, последовательно уменьшая индекс, и возвращает индекс первого найденного элемента. Если указан аргумент *начало*, его значение будет использоваться в качестве начальной позиции поиска; иначе поиск начнется с конца массива. Для проверки эквивалентности используется оператор ===. Возвращает индекс первого соответствующего элемента или -1, если соответствие не было найдено.

См. также

`Array.indexOf()`, `String.lastIndexOf()`

Array.length

размер массива

Синтаксис

массив.`length`

Описание

Свойство `length` массива всегда на единицу больше индекса последнего элемента, определенного в массиве. Для традиционных «плотных» массивов, в которых определена непрерывная последовательность элементов и которые начинаются с элемента 0, свойство `length` указывает количество элементов в массиве.

Свойство `length` инициализируется в момент создания массива с помощью метода-конструктора `Array()`. Добавление новых элементов изменяет значение `length`, если в этом возникает необходимость:

```
a = new Array();                    // a.length равно 0
b = new Array(10);                  // b.length равно 10
c = new Array("one", "two", "three"); // c.length равно 3
c[3] = "four";                      // c.length изменяется на 4
c[10] = "blastoff";                 // c.length становится равным 11
```

Чтобы изменить размер массива, можно установить значение свойства `length`. Если новое значение `length` меньше предыдущего, массив обрезается и элементы в его конце теряются. Если значение `length` увеличивается (новое значение больше старого), массив становится больше, а новые элементы, добавленные в конец массива, получают значение `undefined`.

Array.map() **ECMAScript 5**

вычисляет элементы нового массива из элементов старого массива

Синтаксис

массив.`map(`*f*`)`

массив.`map(`*f*, *o*`)`

Аргументы

f	Функция, вызываемая для каждого элемента *массива*. Возвращаемое ею значение становится элементом возвращаемого массива.
o	Необязательное значение `this`, передаваемое функции *f*.

Возвращаемое значение

Новый массив, элементы которого были вычислены функцией *f*.

Описание

Метод `map()` создает новый массив той же длины, что и прежний *массив*, и вычисляет элементы этого нового массива, передавая элементы *массива* функции *f*. Метод `map()` выполняет цикл по индексам *массива* в порядке их возрастания и вызывает *f* для каждого элемента. Для каждого индекса `i` функция *f* вызывается с тремя аргументам, а ее возвращаемое значение сохраняется в элементе с индексом `i` вновь созданного массива:

```
a[i] = f(array[i], i, array)
```

После того как метод `map()` передаст каждый элемент *массива* функции *f* и сохранит результаты в новом массиве, он вернет новый массив.

Дополнительные сведения приводятся в статье `Array.forEach()`.

Пример

```
[1,2,3].map(function(x) { return x*x; }); // => [1,4,9]
```

См. также

`Array.every()`, `Array.filter()`, `Array.forEach()`, `Array.indexOf()`, `Array.reduce()`

Array.pop()

удаляет и возвращает последний элемент массива

Синтаксис

массив.`pop()`

Возвращаемое значение

Последний элемент *массива*.

Описание

Метод `pop()` удаляет последний элемент *массива*, уменьшает длину массива на единицу и возвращает значение удаленного элемента. Если массив уже пуст, `pop()` его не изменяет и возвращает значение `undefined`.

Пример

Метод `pop()` и парный ему метод `push()` позволяют реализовать стек, работающий по принципу «первым вошел, последним вышел». Например:

```
var stack = [];       // stack: []
stack.push(1, 2);     // stack: [1,2] Вернет 2
stack.pop();          // stack: [1] Вернет 2
stack.push([4,5]);    // stack: [1,[4,5]] Вернет 2
stack.pop()           // stack: [1] Вернет [4,5]
stack.pop();          // stack: [] Вернет 1
```

См. также

`Array.push()`

Array.push()

добавляет элементы массива

Синтаксис

массив.`push(`*значение*, ...`)`

Аргументы

значение, ...

Одно или более значений, которые должны быть добавлены в конец *массива*.

Возвращаемое значение

Новая длина массива после добавления в него указанных значений.

Описание

Метод `push()` добавляет свои аргументы в указанном порядке в конец массива. Он изменяет существующий *массив*, а не создает новый. Метод `push()` и парный ему метод `pop()` используют массив для реализации стека, работающего по принципу «первым вошел, последним вышел». Пример – в статье `Array.pop()`.

См. также

`Array.pop()`

Array.reduce() ECMAScript 5

вычисляет значение из элементов массива

Синтаксис

массив.`reduce(`*f*`)`

массив.`reduce(`*f*, *начальное_значение*`)`

Аргументы

f Функция, объединяющая два значения (два элемента массива) и возвращающая новое значение «свертки».

начальное_значение

Необязательное начальное значение свертки массива. Если этот аргумент указан, метод `reduce()` будет действовать, как если бы это значение было добавлено в начало массива.

Возвращаемое значение

Значение свертки массива, которое является результатом последнего вызова функции *f*.

Описание

Метод `reduce()` в первом аргументе принимает функцию *f*. Эта функция должна действовать как двухместный оператор: она должна принимать два значения, выполнять над ними некоторую операцию и возвращать результат. Если *массив* имеет `n` элементов, функция *f* будет вызвана методом `reduce()` для свертки элементов в единственное значение `n-1` раз. (Возможно, вы уже знакомы с операцией свертки массивов по другим языкам программирования[1].)

При первом вызове функции *f* передаются два первых элемента *массива*. При каждом следующем вызове функции *f* передаются значение, полученное при предыдущем вызове, и следующий элемент (в порядке возрастания индексов) *массива*. Возвращаемое значение последнего вызова становится возвращаемым значением метода `reduce()`.

---

[1] В других языках программирования эта операция может называться «fold» или «inject». Однако в русскоязычной литературе эти термины переводятся как «свертка». – *Прим. перев.*

Методу `reduce()` может предаваться второй необязательный аргумент с *начальным_значением*. Если *начальное_значение* указано, метод `reduce()` будет действовать так, как если бы значение этого аргумента было вставлено в начало *массива* (в реальности массив не модифицируется). Иными словами, если метод `reduce()` вызывается с двумя аргументами, то *начальное_значение* будет использоваться, как если бы оно было получено ранее в результате вызова функции *f*. В этом случаев при первом вызове функции *f* будут переданы *начальное_значение* и первый элемент *массива*. Когда передается *начальное_значение*, создается свертка из `n+1` элементов (`n` элементов *массива* плюс *начальное_значение*) и функция *f* будет вызвана `n` раз.

Если *массив* пуст и *начальное_значение* не указано, метод `reduce()` возбудит исключение `TypeError`. Если *массив* пуст и *начальное_значение* указано, метод `reduce()` вернет *начальное_значение*, не вызвав *f* ни разу. Если массив имеет единственный элемент и *начальное_значение* не указано, метод `reduce()` вернет единственный элемент *массива*, не вызывая функцию *f*.

Выше говорится о двух аргументах функции *f*, но в действительности метод `reduce()` передает этой функции четыре аргумента. В третьем аргументе передается индекс второго аргумента в массиве, а в четвертом – сам *массив*. Функция *f* всегда вызывается как функция, а не как метод.

Пример

```
[1,2,3,4].reduce(function(x,y) { return x*y; }) // => 24: ((1*2)*3)*4
```

См. также

`Array.forEach()`, `Array.map()`, `Array.reduceRight()`

Array.reduceRight() ECMAScript 5

выполняет свертку массива справа налево

Синтаксис

массив.`reduceRight(`*f*`)`

массив.`reduceRight(`*f*, *начальное_значение*`)`

Аргументы

f Функция, объединяющая два значения (два элемента массива) и возвращающая новое значение «свертки».

начальное_значение

Необязательное начальное значение свертки массива. Если этот аргумент указан, метод `reduceRight()` будет действовать, как если бы это значение было добавлено в конец массива.

Возвращаемое значение

Значение свертки массива, которое является результатом последнего вызова функции *f*.

Описание

Метод `reduceRight()` действует подобно методу `reduce()`: он вызывает `n-1` раз функцию *f* для свертки `n` элементов *массива* в единственное значение. Отличие `reduceRight()` от `reduce()` заключается только в том, что он выполняет обход элементов массива справа налево (от больших индексов к меньшим), а не слева направо. Подробности см. в статье `Array.reduce()`.

Пример

```
[2, 10, 60].reduceRight(function(x,y) { return x/y }) // => 3: (60/10)/2
```

См. также

`Array.reduce()`

Array.reverse()

изменяет порядок следования элементов в массиве на противоположный

Синтаксис

массив.`reverse()`

Описание

Метод `reverse()` объекта `Array` меняет порядок следования элементов в массиве на противоположный. Он делает это «на месте», т. е. переупорядочивает элементы указанного *массива*, не создавая новый. Если есть несколько ссылок на *массив*, новый порядок следования элементов массива будет виден по всем ссылкам.

Пример

```
a = new Array(1, 2, 3); // a[0] == 1, a[2] == 3;
a.reverse();            // Теперь a[0] == 3, a[2] == 1;
```

Array.shift()

сдвигает элементы к началу массива

Синтаксис

массив.`shift()`

Возвращаемое значение

Бывший первый элемент массива.

Описание

Метод `shift()` удаляет и возвращает первый элемент массива, смещая все последующие элементы на одну позицию вниз для занятия освободившегося места в начале массива. Если массив пуст, `shift()` не делает ничего и возвращает значение `undefined`. Обратите внимание: `shift()` не создает новый массив, а непосредственно изменяет сам *массив*.

Метод `shift()` похож на `Array.pop()` за исключением того, что удаление элемента производится из начала массива, а не с конца. `shift()` часто используется в сочетании с `unshift()`.

Пример

```
var a = [1, [2,3], 4]
a.shift();  // Вернет 1; a = [[2,3], 4]
a.shift();  // Вернет [2,3]; a = [4]
```

См. также

`Array.pop()`, `Array.unshift()`

Array.slice()

возвращает фрагмент массива

Синтаксис

массив.`slice`(*начало*, *конец*)

Аргументы

начало	Индекс элемента массива, с которого начинается фрагмент. Отрицательное значение этого аргумента указывает позицию, измеряемую от конца массива. Другими словами, -1 обозначает последний элемент, -2 – второй элемент с конца и т. д.
конец	Индекс элемента массива, расположенного непосредственно после конца фрагмента. Если этот аргумент не указан, фрагмент включает все элементы массива от элемента, заданного аргументом *начало*, до конца массива. Если этот аргумент отрицателен, позиция элемента отсчитывается от конца массива.

Возвращаемое значение

Новый массив, содержащий элементы *массива* от элемента, заданного аргументом *начало*, до элемента, определяемого аргументом *конец*, но не включая его.

Описание

Метод `slice()` возвращает фрагмент, или подмассив, *массива*. Возвращаемый массив содержит элемент, заданный аргументом *начало*, и все последующие элементы до элемента, заданного аргументом *конец*, но не включая его. Если аргумент *конец* не указан, возвращаемый массив содержит все элементы от элемента, заданного аргументом *начало*, до конца *массива*.

Обратите внимание: `slice()` не изменяет массив. Для удаления фрагмента массива следует использовать метод `Array.splice()`.

Пример

```
var a = [1,2,3,4,5];
a.slice(0,3);   // Вернет [1,2,3]
a.slice(3);     // Вернет [4,5]
a.slice(1,-1);  // Вернет [2,3,4]
a.slice(-3,-2); // Вернет [3]; в IE 4 работает с ошибкой, возвращая [1,2,3]
```

Ошибки

В Internet Explorer 4 *начало* не может быть отрицательным числом. В более поздних версиях IE эта ошибка исправлена.

См. также

`Array.splice()`

Array.some() ECMAScript 5

проверяет, возвращает ли предикат значение true хотя бы для одного элемента массива

Синтаксис

массив.`some`(*предикат*)

массив.`some`(*предикат*, *o*)

Аргументы

предикат	Функция-предикат для проверки элементов массива.
o	Необязательное значение `this` в вызове функции *предиката*.

Возвращаемое значение

`true`, если *предикат* вернет `true` (или истинное значение) хотя бы для одного элемента *массива*, или `false`, если *предикат* вернет `false` (или ложное значение) для всех элементов.

Описание

Метод `some()` проверяет, выполняется ли условие хотя бы для одного элемента *массива*. Он выполняет цикл по элементам *массива* в порядке возрастания индексов и вызывает указанную функцию *предикат* для каждого элемента. Если *предикат* вернет `true` (или значение, которое в логическом контексте преобразуется в `true`), то метод `some()` прекратит выполнение цикла и немедленно вернет `true`. Если все вызовы *предиката* вернут `false` (или значение, которое в логическом контексте преобразуется в `false`), то метод `some()` вернет `false`. При применении к пустому массиву `some()` вернет `false`.

Этот метод очень похож на метод `every()`. Дополнительные сведения приводятся в статьях `Array.every()` и `Array.forEach()`.

Пример

```
[1,2,3].some(function(x) { return x > 5; }) // => false: нет элементов > 5
[1,2,3].some(function(x) { return x > 2; }) // => true: некоторые > 3
[].some(function(x) { return true; });      // => false: всегда false для []
```

См. также

`Array.every()`, `Array.filter()`, `Array.forEach()`

Array.sort()

сортирует элементы массива

Синтаксис

массив.`sort()`

массив.`sort(`*orderfunc*`)`

Аргументы

orderfunc	Необязательная функция, определяющая порядок сортировки.

Возвращаемое значение

Ссылка на массив. Обратите внимание, что массив сортируется на месте, копия массива не создается.

Описание

Метод `sort()` сортирует элементы *массива* на месте без создания копии массива. Если `sort()` вызывается без аргументов, элементы массива располагаются в алфавитном порядке (точнее, в порядке, определяемом используемой в системе кодировкой символов). Если необходимо, элементы сначала преобразуются в строки, чтобы их можно было сравнивать.

Чтобы отсортировать элементы массива в каком-либо другом порядке, необходимо указать функцию сравнения, которая сравнивает два значения и возвращает число, обозначающее их относительный порядок. Функция сравнения должна принимать два аргумента, *a* и *b*, и возвращать одно из следующих значений:

- Отрицательное число, если в соответствии с выбранным критерием сортировки значение *a* «меньше» значения *b* и должно находиться в отсортированном массиве перед *b*.
- Ноль, если *a* и *b* в смысле сортировки эквивалентны.
- Положительное число, если значение *a* «больше» значения *b*.

Следует отметить, что неопределенные элементы при сортировке всегда оказываются в конце массива. Это происходит, даже если указана специальная функция сортировки: неопределенные значения никогда не передаются в заданную функцию *orderfunc*.

Пример

Следующий фрагмент показывает, как написать функцию сравнения, сортирующую массив чисел в числовом, а не в алфавитном порядке:

```
// Функция сортировки чисел в порядке возрастания
function numberorder(a, b) { return a - b; }
a = new Array(33, 4, 1111, 222);
a.sort();             // Алфавитная сортировка: 1111, 222, 33, 4
a.sort(numberorder); // Числовая сортировка: 4, 33, 222, 1111
```

Array.splice()

вставляет, удаляет или замещает элементы массива

Синтаксис

массив.`splice(`*начало*, *удаляемое_количество*, *значение*, ...`)`

Аргументы

начало

Элемент массива, с которого следует начать вставку или удаление.

удаляемое_количество

Количество элементов, которые должны быть удалены из массива, начиная с элемента, заданного аргументом *начало*, и включая этот элемент. Чтобы выполнить вставку без удаления, в этом аргументе следует передать значение 0.

значение,

Ноль или более значений, которые должны быть вставлены в массив, начиная с индекса, указанного в аргументе *начало*.

Возвращаемое значение

Массив, содержащий удаленные из *массива* элементы, если они есть.

Описание

Метод `splice()` удаляет указанное количество элементов массива, начиная с элемента, позиция которого определяется аргументом *начало*, включая его, и заменяет значениями, перечисленными в списке аргументов. Элементы массива, расположенные после вставляемых или удаляемых элементов, сдвигаются и образуют непрерывную последовательность с остальной частью массива. Однако следует заметить, что, в отличие от метода с похожим именем, `slice()`, метод `splice()` непосредственно изменяет массив.

Пример

Работу `splice()` проще всего понять на примере:

```
var a = [1,2,3,4,5,6,7,8]
a.splice(1,2);      // Вернет [2,3]; а равно [1,4]
a.splice(1,1);      // Вернет [4]; а равно [1]
a.splice(1,0,2,3); // Вернет []; а равно [1 2 3]
```

См. также

`Array.slice()`

Array.toLocaleString()

преобразует массив в локализованную строку **переопределяет Object.toLocaleString()**

Синтаксис

массив.`toLocaleString()`

Возвращаемое значение

Локализованное строковое представление *массива*.

Исключения

`TypeError`	Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод `toLocaleString()` массива возвращает локализованное строковое представление массива. Это делается путем вызова метода `toLocaleString()` для всех элементов массива и последующей конкатенации полученных строк с использованием символа-разделителя, определяемого региональными параметрами настройки.

См. также

`Array.toString()`, `Object.toLocaleString()`

Array.toString()

преобразует массив в строку **переопределяет Object.toString()**

Синтаксис

массив.`toString()`

Возвращаемое значение

Строковое представление *массива*.

Исключения

`TypeError`	Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод `toString()` массива преобразует массив в строку и возвращает эту строку. Когда массив используется в строковом контексте, JavaScript автоматически преобразует его в строку путем вызова этого метода. Однако в некоторых случаях может потребоваться явный вызов `toString()`.

`toString()` сначала преобразует в строку каждый элемент (вызывая их методы `toString()`). После преобразования все элементы выводятся в виде списка строк, разделенных запятыми. Это значение совпадает со значением, возвращаемым методом `join()` без аргументов.

См. также

`Array.toLocaleString()`, `Object.toString()`

Array.unshift()

вставляет элементы в начало массива

Синтаксис

массив.`unshift`(*значение*, ...)

Аргументы

значение,	Одно и более значений, которые должны быть вставлены в начало *массива*.

Возвращаемое значение

Новая длина массива.

Описание

Метод `unshift()` вставляет свои аргументы в начало массива, сдвигая существующие элементы к верхним индексам для освобождения места. Первый аргумент `unshift()` становится новым нулевым элементом массива, второй аргумент – новым первым элементом и т. д. Обратите внимание: `unshift()` не создает новый массив, а изменяет существующий.

Пример

Метод `unshift()` часто используется совместно с `shift()`. Например:

```
var a = [];             // a:[]
a.unshift(1);           // a:[1]          Вернет: 1
a.unshift(22);          // a:[22,1]       Вернет: 2
a.shift();              // a:[1]          Вернет: 22
a.unshift(33,[4,5]);    // a:[33,[4,5],1] Вернет: 3
```

См. также

`Array.shift()`

Boolean

поддержка логических значений **Object→Boolean**

Конструктор

```
new Boolean(значение) // Функция-конструктор
Boolean(значение)     // Функция преобразования
```

Аргументы

значение	Значение, которое должно быть сохранено в объекте Boolean или преобразовано в логическое значение.

Возвращаемое значение

При вызове в качестве конструктора (с оператором `new`) `Boolean()` преобразует аргумент в логическое значение и возвращает объект `Boolean`, содержащий это значение. При вызове в качестве функции (без оператора `new`) `Boolean()` просто преобразует свой аргумент в элементарное логическое значение и возвращает его.

Значения `0`, `NaN`, `null`, пустая строка `""` и значение `undefined` преобразуются в `false`. Все остальные элементарные значения, за исключением `false` (но включая строку `"false"`), а также все объекты и массивы преобразуются в `true`.

Методы

`toString()`	Возвращает `"true"` или `"false"` в зависимости от логического значения, представляемого объектом `Boolean`.
`valueOf()`	Возвращает элементарное логическое значение, содержащееся в объекте `Boolean`.

Описание

Логические значения – это базовый тип данных JavaScript. Объект `Boolean` представляет собой «обертку» вокруг логического значения. Объектный тип `Boolean` в основном существует для предоставления метода `toString()`, который преобразует логические значения в строки. Когда метод `toString()` вызывается для преобразования логического значения в строку (а он часто вызывается интерпретатором JavaScript неявно), логическое значение преобразуется во временный объект `Boolean`, для которого может быть вызван метод `toString()`.

См. также

`Object`

Boolean.toString()

преобразует логическое значение в строку **переопределяет Object.toString()**

Синтаксис

b.`toString()`

Возвращаемое значение

Строка `"true"` или `"false"` в зависимости от того, чем является *b*: элементарным логическим значением или объектом `Boolean`.

Исключения

`TypeError`	Если метод вызывается для объекта, не являющегося объектом `Boolean`.

Boolean.valueOf()

логическое значение объекта Boolean **переопределяет Object.valueOf()**

Синтаксис

b.`valueOf()`

Возвращаемое значение

Элементарное логическое значение, которое содержится в *b*, который является объектом `Boolean`.

Исключения

`TypeError`	Если метод вызывается для объекта, не являющегося `Boolean`.

Date

работа с датами и временем **Object→Date**

Конструктор

new Date()
new Date(*миллисекунды*)
new Date(*строка_даты*)
new Date(*год*, *месяц*, *день*, *часы*, *минуты*, *секунды*, *мс*)

Конструктор `Date()` без аргументов создает объект `Date` со значением, равным текущим дате и времени. Если конструктору передается единственный числовой аргумент, он используется как внутреннее числовое представление даты в миллисекундах, аналогичное значению, возвращаемому методом `getTime()`. Когда передается один строковый аргумент, он рассматривается как строковое представление даты в формате, принимаемом методом `Date.parse()`. Кроме того, конструктору можно передать от двух до семи числовых аргументов, задающих индивидуальные поля даты и времени. Все аргументы, кроме первых двух – полей года и месяца, – могут отсутствовать. Обратите внимание: эти поля даты и времени задаются на основе локального времени, а не времени UTC (Universal Coordinated Time – универсальное скоординированное время), аналогичного GMT (Greenwich Mean Time – среднее время по Гринвичу). В качестве альтернативы может использоваться статический метод `Date.UTC()`.

`Date()` может также вызываться как функция (без оператора `new`). При таком вызове `Date()` игнорирует любые переданные аргументы и возвращает текущие дату и время.

Аргументы

миллисекунды	Количество миллисекунд между нужной датой и полночью 1 января 1970 года (UTC). Например, передав в качестве аргумента число 5000, мы создадим дату, обозначающую пять секунд после полуночи 1 января 1970 года.
строка_даты	Единственный аргумент, задающий дату и (необязательно) время в виде строки. Строка должна иметь формат, понятный для `Date.parse()`.
год	Год в виде четырех цифр. Например, `2001` для 2001 года. Для совместимости с более ранними реализациями JavaScript к аргументу добавляется 1900, если значение аргумента находится между 0 и 99.
месяц	Месяц, заданный в виде целого от 0 (январь) до 11 (декабрь).
день	День месяца, заданный в виде целого от 1 до 31. Обратите внимание, что наименьшее из значений этого аргумента равно 1, а остальных аргументов – 0. Необязательный аргумент.
часы	Часы, заданные в виде целого от 0 (полночь) до 23 (11 часов вечера). Необязательный аргумент.
минуты	Минуты в часах, указанные в виде целого от 0 до 59. Необязательный аргумент.
секунды	Секунды в минутах, указанные в виде целого от 0 до 59. Необязательный аргумент.
мс	Миллисекунды в секунде, указанные в виде целого от 0 до 999. Необязательный аргумент.

Методы

У объекта `Date` нет доступных для записи или чтения свойств; вместо этого доступ к значениям даты и времени выполняется через методы. Большинство методов объекта `Date` имеют две формы: одна для работы с локальным временем, другая – с универсальным временем (UTC или GMT). Если в имени метода присутствует строка «UTC», он работает с универсальным временем. Эти пары методов указываются в приведенном далее списке вместе. Например, обозначение `get[UTC]Day()` относится к двум методам: `getDay()` и `getUTCDay()`.

Методы объекта `Date` могут вызываться только для объектов типа `Date` и генерируют исключение `TypeError`, если вызывать их для объектов другого типа.

`get[UTC]Date()`

Возвращает день месяца из объекта Date в соответствии с локальным или универсальным временем.

`get[UTC]Day()`

Возвращает день недели из объекта `Date` в соответствии с локальным или универсальным временем.

`get[UTC]FullYear()`

Возвращает год даты в полном четырехзначном формате в локальном или универсальном времени.

`get[UTC]Hours()`

Возвращает поле часов в объекте `Date` в локальном или универсальном времени.

`get[UTC]Milliseconds()`

Возвращает поле миллисекунд в объекте Date в локальном или универсальном времени.

`get[UTC]Minutes()`

Возвращает поле минут в объекте `Date` в локальном или универсальном времени.

`get[UTC]Month()`

Возвращает поле месяца в объекте `Date` в локальном или универсальном времени.

`get[UTC]Seconds()`

Возвращает поле секунд в объекте `Date` в локальном или универсальном времени.

`getTime()`

Возвращает внутреннее представление (миллисекунды) объекта `Date`. Обратите внимание: это значение не зависит от часового пояса, следовательно, отдельный метод `getUTCTime()` не нужен.

`getTimezoneOffset()`

Возвращает разницу в минутах между локальным и универсальным представлениями даты. Обратите внимание: возвращаемое значение зависит от того, действует ли для указанной даты летнее время.

`getYear()`

Возвращает поле года в объекте `Date`. Признан устаревшим, рекомендуется вместо него применять метод `getFullYear()`.

`set[UTC]Date()`

Устанавливает день месяца в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]FullYear()`

Устанавливает год (и, возможно, месяц и день) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Hours()`

Устанавливает час (и, возможно, поля минут, секунд и миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Milliseconds()`

Устанавливает поле миллисекунд в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Minutes()`

Устанавливает поле минут (и, возможно, поля секунд и миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Month()`

Устанавливает поле месяца (и, возможно, дня месяца) в `Date` в соответствии с локальным или универсальным временем.

`set[UTC]Seconds()`

Устанавливает поле секунд (и, возможно, поле миллисекунд) в `Date` в соответствии с локальным или универсальным временем.

`setTime()`

Устанавливает поля объекта `Date` в соответствии с миллисекундным форматом.

`setYear()`

Устанавливает поле года объекта `Date`. Признан устаревшим, вместо него рекомендуется использовать `setFullYear()`.

`toDateString()`

Возвращает строку, представляющую дату из `Date` для локального часового пояса.

`toGMTString()`

Преобразует `Date` в строку, беря за основу часовой пояс GMT. Признан устаревшим, вместо него рекомендован метод `toUTCString()`.

`toISOString()`

Преобразует `Date` в строку, используя стандарт ISO-8601, объединяющий формат представления даты/времени и UTC.

`toJSON()`

Сериализует объект `Date` в формат JSON с помощью метода `toISOString()`.

`toLocaleDateString()`

Возвращает строку, представляющую дату из `Date` в локальном часовом поясе в соответствии с локальными соглашениями по форматированию дат.

`toLocaleString()`

Преобразует `Date` в строку в соответствии с локальным часовым поясом и локальными соглашениями о форматировании дат.

`toLocaleTimeString()`

Возвращает строку, представляющую время из `Date` в локальном часовом поясе на основе локальных соглашений о форматировании времени.

`toString()`

Преобразует `Date` в строку в соответствии с локальным часовым поясом.

`toTimeString()`	Возвращает строку, представляющую время из `Date` в локальном часовом поясе.
`toUTCString()`	Преобразует `Date` в строку, используя универсальное время.
`valueOf()`	Преобразует объект `Date` в его внутренний миллисекундный формат.

Статические методы

В дополнение к перечисленным методам экземпляра в объекте `Date` определены два статических метода. Эти методы вызываются через сам конструктор `Date()`, а не через отдельные объекты `Date`:

`Date.now()`	Возвращает текущее время в миллисекундах с начала эпохи.
`Date.parse()`	Анализирует строковое представление даты и времени и возвращает внутреннее представление этой даты в миллисекундах.
`Date.UTC()`	Возвращает представление указанной даты и времени UTC в миллисекундах.

Описание

Объект `Date` – это тип данных, встроенный в язык JavaScript. Объекты `Date` создаются с помощью представленного ранее синтаксиса `new Date()`.

После создания объекта `Date` можно воспользоваться его многочисленными методами. Многие из методов позволяют получать и устанавливать поля года, месяца, дня, часа, минуты, секунды и миллисекунды в соответствии либо с локальным временем, либо с временем UTC (универсальным, или GMT). Метод `toString()` и его варианты преобразуют даты в понятные для восприятия строки. `getTime()` и `setTime()` преобразуют количество миллисекунд, прошедших с полуночи (GMT) 1 января 1970 года, во внутреннее представление объекта `Date` и обратно. В этом стандартном миллисекундном формате дата и время представляются одним целым, что делает дату очень простой арифметически. Стандарт ECMAScript требует, чтобы объект `Date` мог представить любые дату и время с миллисекундной точностью в пределах 100 миллионов дней до и после 01.01.1970. Этот диапазон равен ±273 785 лет, поэтому JavaScript-часы будут правильно работать до 275 755 года.

Пример

Известно множество методов, позволяющих работать с созданным объектом `Date`:

```
d = new Date(); // Получает текущую дату и время
document.write('Сегодня: "+d.toLocaleDateString() + '. ');// Показывает дату
document.write('Время: '+ d.toLocaleTimeString());       // Показывает время
var dayOfWeek = d.getDay();                               // День недели
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6);       // Сегодня выходной?
```

Еще одно типичное применение объекта `Date` – это вычитание миллисекундного представления текущего времени из другого времени для определения относительного местоположения двух временных меток. Следующий пример клиентского кода показывает два таких применения:

```
<script language="JavaScript">
today = new Date();     // Запомнить сегодняшнюю дату
christmas = new Date(); // Получить дату из текущего года
christmas.setMonth(11); // Установить месяц декабрь...
christmas.setDate(25);  // и 25-е число
// Если Рождество еще не прошло, вычислить количество миллисекунд между текущим моментом
// и Рождеством, преобразовать его в количество дней и вывести сообщение
```

```
if (today.getTime() < christmas.getTime()) {
    difference = christmas.getTime() - today.getTime();
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
    document.write('Всего ' + difference + ' дней до Рождества!<p>');
}
</script>
// ... остальная часть HTML-документа ...
<script language="JavaScript">
// Здесь мы используем объекты Date для измерения времени
// Делим на 1000 для преобразования миллисекунд в секунды
now = new Date();
document.write('<p>Страница загружалась' +
    (now.getTime()-today.getTime())/1000 +
    'секунд.');
</script>
```

См. также

`Date.parse()`, `Date.UTC()`

Date.getDate()

возвращает значение поля дня месяца объекта Date

Синтаксис

дата.`getDate()`

Возвращаемое значение

День месяца *даты*, представляющей собой объект `Date`, в соответствии с локальным временем. Возвращаемые значения могут находиться в интервале между 1 и 31.

Date.getDay()

возвращает значение поля дня недели объекта Date

Синтаксис

дата.`getDay()`

Возвращаемое значение

День недели *даты*, представляющей собой объект `Date`, в соответствии с локальным временем. Возвращает числа от 0 (воскресенье) до 6 (суббота).

Date.getFullYear()

возвращает значение поля года объекта Date

Синтаксис

дата.`getFullYear()`

Возвращаемое значение

Год, получаемый, когда *дата* выражена в локальном времени. Возвращает четыре цифры, а не сокращение из двух цифр.

Date.getHours()

возвращает значение поля часа объекта Date

Синтаксис

дата.`getHours()`

Возвращаемое значение

Значение поля часа в *дате*, представляющей собой объект `Date`, в локальном времени. Возвращаемое значение находится в диапазоне между 0 (полночь) и 23 (11 часов вечера).

Date.getMilliseconds()

возвращает значение поля миллисекунд объекта Date

Синтаксис

дата.`getMilliseconds()`

Возвращаемое значение

Поле миллисекунд в *дате*, представляющей собой объект `Date`, вычисленное в локальном времени.

Date.getMinutes()

возвращает значение поля минут объекта Date

Синтаксис

дата.`getMinutes()`

Возвращаемое значение

Поле минут в *дате*, представляющей собой объект `Date`, вычисленное в локальном времени. Возвращаемое значение может принимать значения от 0 до 59.

Date.getMonth()

возвращает значение поля месяца объекта Date

Синтаксис

дата.`getMonth()`

Возвращаемое значение

Поле месяца в *дате*, представляющей собой объект `Date`, вычисленное в локальном времени. Возвращаемое значение может принимать значения от 0 (январь) до 11 (декабрь).

Date.getSeconds()

возвращает значение поля секунд объекта Date

Синтаксис

дата.`getSeconds()`

Возвращаемое значение

Поле секунд в *дате*, представляющем собой объект `Date`, в локальном времени. Возвращаемое значение может принимать значения от 0 до 59.

Date.getTime()

возвращает значение даты в миллисекундах

Синтаксис

дата.`getTime()`

Возвращаемое значение

Миллисекундное представление *даты*, представляющей собой объект `Date`, т.е. число миллисекунд между полночью 01.01.1970 и датой/временем, определяемыми *датой*.

Описание

Метод `getTime()` преобразует дату и время в одно целое значение. Это удобно, когда требуется сравнить два объекта `Date` или определить время, прошедшее между двумя датами. Обратите внимание: миллисекундное представление даты не зависит от часового пояса, поэтому отсутствует метод `getUTCTime()`, дополняющий данный. Не путайте метод `getTime()` с методами `getDay()` и `getDate()`, возвращающими, соответственно, день недели и день месяца.

Методы `Date.parse()` и `Date.UTC()` позволяют преобразовать спецификацию даты и времени в миллисекундное представление, обходя избыточное создание объекта `Date`.

См. также

`Date`, `Date.parse()`, `Date.setTime()`, `Date.UTC()`

Date.getTimezoneOffset()

определяет смещение относительно GMT

Синтаксис

дата.`getTimezoneOffset()`

Возвращаемое значение

Разница в минутах между временем по Гринвичу (GMT) и локальным временем.

Описание

Функция `getTimezoneOffset()` возвращает разницу в минутах между универсальным и локальным временем, сообщая, в каком часовом поясе выполняется JavaScript-код и действует ли (или будет ли действовать) летнее время для указанной *даты*.

Возвращаемое значение измеряется в минутах, а не в часах, поскольку в некоторых странах имеются часовые пояса, не занимающие целого часового интервала.

Date.getUTCDate()

возвращает значение поля дня месяца объекта Date (универсальное время)

Синтаксис

дата.`getUTCDate()`

Возвращаемое значение

День месяца (значение между 1 и 31), полученный при вычислении *даты* в универсальном времени.

Date.getUTCDay()

возвращает значение поля дня недели объекта Date (универсальное время)

Синтаксис

дата.`getUTCDay()`

Возвращаемое значение

День недели, получаемый, когда *дата* выражена в универсальном времени. Возвращаемые значения могут находиться в интервале между 0 (воскресенье) и 6 (суббота).

Date.getUTCFullYear()

возвращает значение поля года объекта Date (универсальное время)

Синтаксис

дата.`getUTCFullYear()`

Возвращаемое значение

Год, получаемый, когда *дата* вычисляется в универсальном времени. Возвращаемое значение – четырехзначный номер года, а не сокращение из двух цифр.

Date.getUTCHours()

возвращает значение поля часов объекта Date (универсальное время)

Синтаксис

дата.`getUTCHours()`

Возвращаемое значение

Поле часов для *даты*, вычисленное в универсальном времени. Возвращаемое значение – целое между 0 (полночь) и 23 (11 часов вечера).

Date.getUTCMilliseconds()

возвращает значение поля миллисекунд объекта Date (универсальное время)

Синтаксис

дата.`getUTCMilliseconds()`

Возвращаемое значение

Поле миллисекунд для *даты*, выраженное в универсальном времени.

Date.getUTCMinutes()

возвращает значение поля минут объекта Date (универсальное время)

Синтаксис

дата.`getUTCMinutes()`

Возвращаемое значение

Поле минут для *даты*, вычисленное в универсальном времени. Возвращает целое между 0 и 59.

Date.getUTCMonth()

возвращает значение поля месяца года объекта Date (универсальное время)

Синтаксис

дата.`getUTCMonth()`

Возвращаемое значение

Месяц года, получающийся, когда *дата* вычислена в универсальном времени. Возвращает целое между 0 (январь) и 11 (декабрь). Обратите внимание: объект `Date` обозначает первый день месяца цифрой 1, но первому месяцу года соответствует цифра 0.

Date.getUTCSeconds()

возвращает значение поля секунд объекта Date (универсальное время)

Синтаксис

`дата.getUTCSeconds()`

Возвращаемое значение

Поле секунд *даты*, вычисленное в универсальном времени. Возвращает целое между 0 и 59.

Date.getYear() устарел

возвращает значение поля года объекта Date

Синтаксис

дата.`getYear()`

Возвращаемое значение

Поле года для указанного *даты*, представляющей собой объект `Date`, минус 1900.

Описание

Метод `getYear()` возвращает поле года для указанного объекта `Date` минус 1900. Согласно спецификации ECMAScript v3, этот метод не является обязательным в совместимых реализациях JavaScript; используйте вместо него метод `getFullYear()`.

Date.now() ECMAScript 5

возвращает текущее время в миллисекундах

Синтаксис

`Date.now()`

Возвращаемое значение

Текущее время в миллисекундах, прошедшее с полуночи 1 января 1970 года по Гринвичу.

Описание

До выхода спецификации ECMAScript 5 этот метод можно было реализовать следующим образом:

```
Date.now = function() { return (new Date()).getTime(); }
```

См. также

`Date`, `Date.getTime()`

Date.parse()

синтаксический разбор строки даты/времени

Синтаксис

`Date.parse(`*дата*`)`

Аргументы

дата	Строка для разбора, содержащая дату и время.

Возвращаемое значение

Количество миллисекунд между указанными датой/временем и полночью 1 января 1970 года по Гринвичу.

Описание

Метод `Date.parse()` – это статический метод объекта `Date`. Метод `Date.parse()` принимает один строковый аргумент, анализирует дату, содержащуюся в строке, и возвращает ее в виде числа миллисекунд, прошедших с начала эпохи. Это возвращаемое значение может использоваться непосредственно для создания нового объекта `Date` или для установки даты в существующем объекте `Date` с помощью `Date.setTime()`.

Стандарт ECMAScript 5 требует, чтобы этот метод мог разбирать строки, возвращаемые методом `Date.toISOString()`. В ECMAScript 5 и более ранних версиях спецификации требовалось также, чтобы этот метод мог разбирать строки, возвращаемые методами `toString()` и `toUTCString()`.

См. также

`Date`, `Date.setTime()`, `Date.toISOString()`, `Date.toString()`

Date.setDate()

устанавливает поле дня месяца объекта Date

Синтаксис

дата`.setDate(`*день_месяца*`)`

Аргументы

день_месяца	Целое между 1 и 31, используемое как новое значение (в локальном времени) поля дня месяца объекта *дата*.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setFullYear()

устанавливает поля года и, если явно указано, месяца и дня месяца объекта Date

Синтаксис

дата.`setFullYear(`*год*`)`

дата.`setFullYear(`*год*, *месяц*`)`

дата.`setFullYear(`*год*, *месяц*, *день*`)`

Аргументы

год	Год, выраженный в локальном времени, который должен быть установлен в *дате*. Этот аргумент должен быть целым, включающим век, например 1999; не может быть сокращением, таким как 99.
месяц	Необязательное целое между 0 и 11, используемое для установки нового значения поля месяца (в локальном времени) для *даты*.
день	Необязательное целое между 1 и 31, используемое как новое значение поля дня месяца для *даты* (в локальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setHours()

устанавливает значения полей часов, минут, секунд и миллисекунд объекта Date

Синтаксис

дата.`setHours(`*часы*`)`

дата.`setHours(`*часы*, *минуты*`)`

дата.`setHours(`*часы*, *минуты*, *секунды*`)`

дата.`setHours(`*часы*, *минуты*, *секунды*, *миллисекунды*`)`

Аргументы

часы	Целое между 0 (полночь) и 23 (11 часов вечера) локального времени, устанавливаемое в качестве нового значения часов в *дате*.
минуты	Необязательное целое между 0 и 59, используемое в качестве нового значения поля минут в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.
секунды	Необязательное целое между 0 и 59. Представляет собой новое значение поля секунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.
миллисекунды	Необязательное целое между 0 и 999, используемое как новое значение поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setMilliseconds()

устанавливает значение поля миллисекунд объекта Date

Синтаксис

дата.setMilliseconds(*миллисекунды*)

Аргументы

миллисекунды	Поле миллисекунд, выраженное в локальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Миллисекундное представление измененной даты.

Date.setMinutes()

устанавливает значения полей минут, секунд и миллисекунд объекта Date

Синтаксис

дата.setMinutes(*минуты*)

дата.setMinutes(*минуты*, *секунды*)

дата.setMinutes(*минуты*, *секунды*, *миллисекунды*)

Аргументы

минуты	Целое между 0 и 59, устанавливаемое в качестве значения минут (в локальном времени) в *дате*, представляющей собой объект Date.
секунды	Необязательное целое между 0 и 59, используемое как новое значение поля секунд *даты* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.
миллисекунды	Необязательное целое между 0 и 999, представляющее собой новое значение (в локальном времени) поля миллисекунд *даты*. Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setMonth()

устанавливает поля месяца и дня месяца объекта Date

Синтаксис

дата.setMonth(*месяц*)

дата.setMonth(*месяц*, *день*)

Аргументы

месяц	Целое между 0 (январь) и 11 (декабрь), устанавливаемое в качестве значения поля месяца (в локальном времени) в *дате*, представляющей собой объект Date. Обратите внимание: месяцы нумеруются, начиная с 0, а дни в месяце – с 1.

день	Необязательное целое между 1 и 31, используемое как новое значение поля дня месяца в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setSeconds()

устанавливает значения полей секунд и миллисекунд объекта Date

Синтаксис

дата.`setSeconds`(*секунды*)

дата.`setSeconds`(*секунды*, *миллисекунды*)

Аргументы

секунды	Целое между 0 и 59, устанавливаемое как значение секунд в *дате*, представляющей собой объект `Date`.
миллисекунды	Необязательное целое между 0 и 999, используемое как новое значение поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Date.setTime()

устанавливает значение даты в миллисекундах

Синтаксис

дата.`setTime`(*миллисекунды*)

Аргументы

миллисекунды	Количество миллисекунд между требуемыми датой/временем и полночью 1 января 1970 года по Гринвичу. Подобное миллисекундное значение может быть также передано конструктору `Date()` и получено при вызове методов `Date.UTC()` и `Date.parse()`. Представление даты в миллисекундном формате делает ее независимой от часового пояса.

Возвращаемое значение

Аргумент *миллисекунды*. До выхода стандарта ECMAScript метод ничего не возвращал.

Date.setUTCDate()

устанавливает значение поля дня месяца объекта Date (универсальное время)

Синтаксис

дата.`setUTCDate`(*день_месяца*)

Аргументы

день_месяца	День месяца, выраженный в универсальном времени и устанавливаемый в *дате*. Этот аргумент должен быть целым между 1 и 31.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCFullYear()

устанавливает значения полей года, месяца и дня месяца объекта Date (универсальное время)

Синтаксис

дата.setUTCFullYear(*год*)

дата.setUTCFullYear(*год*, *месяц*)

дата.setUTCFullYear(*год*, *месяц*, *день*)

Аргументы

год	Год, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым, включающим век, например 1999, а не сокращением, как 99.
месяц	Необязательное целое между 0 и 11, используемое как новое значение поля месяца даты (в универсальном времени). Обратите внимание: месяцы нумеруются, начиная с 0, тогда как нумерация дней месяцев начинается с 1.
день	Необязательное целое между 1 и 31; используется как новое значение (в универсальном времени) поля «день месяца» в *дате*.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCHours()

устанавливает значения полей часов, минут, секунд и миллисекунд объекта Date (универсальное время)

Синтаксис

дата.setUTCHours(*часы*)

дата.setUTCHours(*часы*, *минуты*)

дата.setUTCHours(*часы*, *минуты*, *секунды*)

дата.setUTCHours(*часы*, *минуты*, *секунды*, *миллисекунды*)

Аргументы

часы	Значение поля часов, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (полночь) и 23 (11 часов вечера).
минуты	Необязательное целое между 0 и 59, используемое как новое значение поля минут в *дате* (в универсальном времени).
секунды	Необязательное целое между 0 и 59, используемое как новое значение поля секунд в *дате* (в универсальном времени).

миллисекунды	Необязательное целое между 0 и 999, используемое как новое значение поля миллисекунд в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMilliseconds()

устанавливает значение поля миллисекунд объекта Date (универсальное время)

Синтаксис

дата.`setUTCMilliseconds`(*миллисекунды*)

Аргументы

миллисекунды	Значение поля миллисекунд, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMinutes()

устанавливает значения полей минут, секунд и миллисекунд объекта Date (универсальное время)

Синтаксис

дата.`setUTCMinutes`(*минуты*)
дата.`setUTCMinutes`(*минуты*, *секунды*)
дата.`setUTCMinutes`(*минуты*, *секунды*, *миллисекунды*)

Аргументы

минуты	Значение поля минут в универсальном времени для установки в *дате*. Этот аргумент должен принимать значение между 0 и 59.
секунды	Необязательное целое между 0 и 59, используемое как новое значение поля секунд в *дате* (в универсальном времени).
миллисекунды	Необязательное целое между 0 и 999, используемое как новое значение поля миллисекунд в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMonth()

устанавливает поля месяца и дня месяца объекта Date (универсальное время)

Синтаксис

дата.`setUTCMonth`(*месяц*)
дата.`setUTCMonth`(*месяц*, *день*)

Аргументы

месяц	Месяц, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (январь) и 11 (декабрь). Обратите внимание: месяцы нумеруются, начиная с 0, а дни в месяце – с 1.

день	Необязательное целое между 1 и 31, используемое как новое значение поля дня месяца в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCSeconds()

устанавливает значения полей секунд и миллисекунд объекта Date (универсальное время)

Синтаксис

дата.`setUTCSeconds`(*секунды*)

дата.`setUTCSeconds`(*секунды*, *миллисекунды*)

Аргументы

секунды	Значение поля секунд в универсальном времени для установки в *дате*. Этот аргумент должен быть целым между 0 и 59.
миллисекунды	Необязательное целое между 0 и 999, используемое как новое значение поля миллисекунд *даты* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setYear()

устарел

устанавливает поле года объекта Date

Синтаксис

дата.`setYear`(*год*)

Аргументы

год	Целое, устанавливаемое как значение поля года (в локальном времени) в *дате*, представляющей собой объект `Date`. Если это значение находится между 0 и 99, к нему добавляется 1900, и оно рассматривается как год между 1900 и 1999.

Возвращаемое значение

Миллисекундное представление измененной даты. До выхода стандарта ECMAScript этот метод ничего не возвращал.

Описание

Метод `setYear()` устанавливает поле года в указанном объекте `Date`, особым образом обрабатывая интервал времени между 1900 и 1999 годами.

Согласно спецификации ECMAScript v3, этот метод не является обязательным в совместимых реализациях JavaScript; вместо него рекомендован метод `setFullYear()`.

Date.toDateString()

возвращает дату из объекта Date в виде строки

Синтаксис

дата.`toDateString()`

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление даты (без времени), представленной объектом *дата* в локальном времени.

См. также

`Date.toString()`, `Date.toTimeString()`

Date.toGMTString() устарел

преобразует объект Date в строку универсального времени

Синтаксис

дата.`toGMTString()`

Возвращаемое значение

Строковое представление даты и времени, указанное в объекте *дата*. Перед преобразованием в строку, дата переводится из локального времени во время по Гринвичу.

Описание

Метод `toGMTString()` признан устаревшим, вместо него рекомендуется использовать аналогичный метод `Date.toUTCString()`.

Согласно спецификации ECMAScript v3 совместимые реализации JavaScript больше не обязаны предоставлять этот метод; используйте вместо него метод `toUTCString()`.

См. также

`Date.toUTCString()`

Date.toISOString() ECMAScript 5

преобразует объект Date в строку в формате ISO8601

Синтаксис

дата.`toISOString()`

Возвращаемое значение

Строковое представление *даты*, отформатированное в соответствии со стандартом ISO-8601 и выраженное как точная комбинация даты и времени в UTC с указанием часового пояса «Z». Возвращаемая строка имеет следующий формат:

```
yyyy-mm-ddThh:mm:ss.sssZ
```

См. также

`Date.parse()`, `Date.toString()`

Date.toJSON() ECMAScript 5

сериализует объект Date в формат JSON

Синтаксис

дата.`toJSON(`*ключ*`)`

Аргументы

ключ	Метод `toJSON()` не использует этот аргумент и просто передает его функции `JSON.stringify()`.

Возвращаемое значение

Строковое представление *даты*, полученное вызовом метода `toISOString()`.

Описание

Для преобразования объекта `Date` в строку этот метод использует функцию `JSON.stringify()`. Он не предназначен для широкого использования.

См. также

`Date.toISOString()`, `JSON.stringify()`

Date.toLocaleDateString()

возвращает дату из Date в виде строки с учетом региональных настроек

Синтаксис

дата.`toLocaleDateString()`

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление даты (без времени) из объекта *дата*, выраженное в локальном времени и отформатированное в соответствии с региональными настройками.

См. также

`Date.toDateString()`, `Date.toLocaleString()`, `Date.toLocaleTimeString()`, `Date.toString()`, `Date.toTimeString()`

Date.toLocaleString()

преобразует дату в строку с учетом региональных настроек

Синтаксис

дата.`toLocaleString()`

Возвращаемое значение

Строковое представление даты и времени в объекте *дата*. Дата и время представлены в локальном часовом поясе и отформатированы в соответствии с региональными настройками.

Порядок использования

Метод `toLocaleString()` преобразует дату в строку в соответствии с локальным часовым поясом. При форматировании даты и времени используются региональные настройки, поэтому формат может отличаться на разных платформах и в разных странах. Метод `toLocaleString()` возвращает строку, отформатированную в соответствии с предпочтительным для пользователя форматом представления даты и времени.

См. также

`Date.toISOString()`, `Date.toLocaleDateString()`, `Date.toLocaleTimeString()`, `Date.toString()`, `Date.toUTCString()`

Date.toLocaleTimeString()

возвращает время из Date в виде строки с учетом региональных настроек

Синтаксис

дата.`toLocaleTimeString()`

Возвращаемое значение

Зависящее от реализации и понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе и отформатированное в соответствии с региональными настройками.

См. также

`Date.toDateString()`, `Date.toLocaleDateString()`, `Date.toLocaleString()`, `Date.toString()`, `Date.toTimeString()`

Date.toString()

преобразует объект Date в строку **переопределяет Object.toString()**

Синтаксис

дата.`toString()`

Возвращаемое значение

Понятное человеку строковое представление *даты* в локальном часовом поясе.

Описание

Метод `toString()` возвращает понятное человеку и зависящее от реализации строковое представление *даты*. В отличие от `toUTCString()`, метод `toString()` вычисляет дату в локальном часовом поясе. В отличие от `toLocaleString()`, метод `toString()` может представлять дату и время без учета региональных настроек.

См. также

`Date.parse()`, `Date.toDateString()`, `Date.toISOString()`, `Date.toLocaleString()`, `Date.toTimeString()`, `Date.toUTCString()`

Date.toTimeString()

возвращает время из объекта Date в виде строки

Синтаксис

дата.`toTimeString()`

Возвращаемое значение

Зависящее от реализации, понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе.

См. также

`Date.toString()`, `Date.toDateString()`, `Date.toLocaleTimeString()`

Date.toUTCString()

преобразует объект Date в строку (универсальное время)

Синтаксис

дата.`toUTCString()`

Возвращаемое значение

Понятное человеку строковое представление даты, выраженное в универсальном времени.

Описание

Метод `toUTCString()` возвращает зависящую от реализации строку, представляющую дату в универсальном времени.

См. также

`Date.toISOString()`, `Date.toLocaleString()`, `Date.toString()`

Date.UTC()

преобразует спецификацию даты в миллисекунды

Синтаксис

`Date.UTC(`*год*, *месяц*, *день*, *часы*, *минуты*, *секунды*, *мс*`)`

Аргументы

год	Год в четырехзначном формате. Если аргумент находится между 0 и 99, к нему добавляется 1900, и он рассматривается как год между 1900 и 1999.
месяц	Месяц в виде целого числа от 0 (январь) до 11 (декабрь).
день	День месяца в виде целого числа от 1 до 31. Обратите внимание: наименьшее значение этого аргумента равно 1, наименьшее значение других аргументов – 0. Этот аргумент не является обязательным.
часы	Час в виде целого числа от 0 (полночь) до 23 (11 часов вечера). Этот аргумент может отсутствовать.
минуты	Минуты в часе в виде целого числа от 0 до 59. Этот аргумент может отсутствовать.
секунды	Секунды в минутах в виде целого числа от 0 до 59. Этот аргумент может отсутствовать.
мс	Количество миллисекунд. Этот аргумент может отсутствовать; игнорировался до выхода стандарта ECMAScript.

Возвращаемое значение

Миллисекундное представление указанного универсального времени. Метод возвращает количество миллисекунд между полночью по Гринвичу 1 января 1970 года и указанным временем.

Описание

Метод `Date.UTC()` – это статический метод, который вызывается через конструктор `Date()`, а не через отдельный объект `Date`.

Аргументы `Date.UTC()` определяют дату и время и подразумевают время в формате UTC. Указанное время UTC преобразуется в миллисекундный формат, который может использоваться методом-конструктором `Date()` и методом `Date.setTime()`.

Метод-конструктор `Date()` может принимать аргументы даты и времени, идентичные тем, что принимает метод `Date.UTC()`. Разница в том, что конструктор `Date()` подразумевает локальное время, а `Date.UTC()` – время по Гринвичу (GMT). Создать объект Date, используя спецификацию времени в UTC, можно следующим образом:

```
d = new Date(Date.UTC(1996, 4, 8, 16, 30));
```

См. также

`Date`, `Date.parse()`, `Date.setTime()`

Date.valueOf()

преобразует объект Date в миллисекунды **переопределяет Object.valueOf()**

Синтаксис

дата.`valueOf()`

Возвращаемое значение

Миллисекундное представление *даты*. Возвращаемое значение совпадает со значением, возвращаемым `Date.getTime()`.

decodeURI()

декодирует символы в URI

Синтаксис

`decodeURI(`*uri*`)`

Аргументы

uri	Строка, содержащая в закодированном виде URI (Uniform Resource Identifier – унифицированный идентификатор ресурса) или другой текст, подлежащий декодированию.

Возвращаемое значение

Копия аргумента *uri*, в которой все шестнадцатеричные управляющие последовательности заменены на символы, которые они представляют.

Исключения

`URIError`	Означает, что одна или несколько управляющих последовательностей в *uri* имеют неверный формат и не могут быть правильно декодированы.

Описание

`decodeURI()` – это глобальная функция, возвращающая декодированную копию аргумента *uri*. Она выполняет действие, обратное действию функции `encodeURI()`; подробности см. в описании этой функции.

См. также

`decodeURIComponent()`, `encodeURI()`, `encodeURIComponent()`, `escape()`, `unescape()`

decodeURIComponent()

декодирует управляющие последовательности символов в компоненте URI

Синтаксис

`decodeURIComponent(s)`

Аргументы

s Строка, содержащая закодированный компонент URI или другой текст, который должен быть декодирован.

Возвращаемое значение

Копия аргумента *s*, в которой шестнадцатеричные управляющие последовательности заменены представляемыми ими символами.

Исключения

`URIError` Означает, что одна или несколько управляющих последовательностей в аргументе *s* имеют неверный формат и не могут быть правильно декодированы.

Описание

`decodeURIComponent()` – глобальная функция, возвращающая декодированную копию своего аргумента *s*. Ее действие обратно кодированию, выполняемому функцией `encodeURIComponent()`; подробности см. в справочной статье по этой функции.

См. также

`decodeURI()`, `encodeURI()`, `encodeURIComponent()`, `escape()`, `unescape()`

encodeURI()

выполняет кодирование URI с помощью управляющих последовательностей

Синтаксис

`encodeURI(uri)`

Аргументы

uri Строка, содержащая URI или другой текст, который должен быть закодирован.

Возвращаемое значение

Копия аргумента *uri*, в которой некоторые символы заменены шестнадцатеричными управляющими последовательностями.

Исключения

`URIError` Указывает, что строка *uri* содержит искаженные суррогатные пары символов Юникода и не может быть закодирована.

Описание

`encodeURI()` – это глобальная функция, возвращающая закодированную копию аргумента *uri*. Не кодируются символы, цифры и следующие знаки пунктуации ASCII:

```
- _ . ! ~ * ' ( )
```

Функция `encodeURI()` кодирует URI целиком, поэтому следующие символы пунктуации, имеющие в URI специальное значение, также не кодируются:

```
; / ? : @ & = + $ , #
```

Любые другие символы в *uri* заменяются путем преобразования символа в его код UTF-8 и последующего кодирования каждого из полученных байтов шестнадцатеричной управляющей последовательностью в формате `%xx`. В этой схеме кодирования ASCII-символы заменяются одной последовательностью `%xx`, символы с кодами от `\u0080` до `\u07ff` – двумя управляющими последовательностями, а все остальные 16-разрядные символы Юникода – тремя управляющими последовательностями.

При использовании этого метода для кодирования URI необходимо быть уверенным, что ни один из компонентов URI (например, строка запроса) не содержит символов-разделителей URI, таких как `?` и `#`. Если компоненты могут содержать эти символы, необходимо кодировать каждый компонент отдельно с помощью функции `encodeURIComponent()`.

Метод `decodeURI()` предназначен для выполнения действия, обратного кодированию. До выхода ECMAScript v3 с помощью методов `escape()` и `unescape()`, сейчас признанных устаревшими, выполнялись сходные кодирование и декодирование.

Пример

```
// Возвращает http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURI("http://www.isp.com/app.cgi?arg1=1&arg2=hello world");
encodeURI("\u00a9"); // Символ копирайта кодируется в %C2%A9
```

См. также

`decodeURI()`, `decodeURIComponent()`, `encodeURIComponent()`, `escape()`, `unescape()`

encodeURIComponent()

выполняет кодирование компонентов URI с помощью управляющих последовательностей

Синтаксис

`encodeURIComponent(s)`

Аргументы

s	Строка, содержащая фрагмент URI или другой текст, подлежащий кодированию.

Возвращаемое значение

Копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Исключения

`URIError`	Указывает, что строка *s* содержит искаженные суррогатные пары символов Юникода и не может быть закодирована.

Описание

`encodeURIComponent()` – это глобальная функция, возвращающая закодированную копию своего аргумента *s*. Не кодируются буквы, цифры и следующие знаки пунктуации ASCII:

```
- _ . ! ~ * ' ( )
```

Все остальные символы, в том числе такие символы пунктуации, как `/ : #`, служащие для разделения различных компонентов URI, заменяются одной или несколькими шестнадцатеричными управляющими последовательностями. Описание используемой схемы кодирования см. в статье, посвященной функции `encodeURI()`.

Обратите внимание на разницу между `encodeURIComponent()` и `encodeURI()`: функция `encodeURIComponent()` предполагает, что ее аргументом является фрагмент URI (такой как протокол, имя хоста, путь или строка запроса). Поэтому она преобразует символы пунктуации, используемые для разделения фрагментов URI.

Пример

```
encodeURIComponent("hello world?"); // Вернет hello%20world%3F
```

См. также

`decodeURI()`, `decodeURIComponent()`, `encodeURI()`, `escape()`, `unescape()`

Error

обобщенное исключение **Object→Error**

Конструктор

`new Error()`

`new Error(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке с дополнительной информацией об исключении.

Возвращаемое значение

Вновь созданный объект `Error`. Если задан аргумент *сообщение*, объект `Error` будет использовать его в качестве значения своего свойства `message`; в противном случае он возьмет в качестве значения этого свойства строку по умолчанию, определенную реализацией. Когда конструктор `Error()` вызывается как функция (без оператора `new`), он ведет себя так же, как при вызове с оператором `new`.

Свойства

`message`	Сообщение об ошибке с дополнительной информацией об исключении. В этом свойстве хранится строка, переданная конструктору, или строка по умолчанию, определяемая реализацией.
`name`	Строка, задающая тип исключения. Для экземпляров класса `Error` и всех его подклассов это свойство задает имя конструктора, с помощью которого был создан экземпляр.

Методы

`toString()`	Возвращает строку, определенную в реализации, которая представляет этот объект `Error`.

Описание

Экземпляры класса `Error` представляют ошибки или исключения и обычно используются с инструкциями `throw` и `try/catch`. Свойство `name` определяет тип исключения, а посредством свойства `message` можно создать и отправить пользователю сообщение с подробной информацией об исключении.

Интерпретатор JavaScript никогда не создает объект `Error` непосредственно. Вместо этого он создает экземпляры одного из подклассов `Error`, таких как `SyntaxError` или

`RangeError`. В ваших программах для предупреждения об исключении может быть удобнее создавать объекты `Error` или просто выдавать сообщение об ошибке или ее код в виде элементарного строкового или числового значения.

Обратите внимание: спецификация ECMAScript определяет для класса `Error` метод `toString()` (он наследуется всеми подклассами `Error`), но не требует, чтобы этот метод возвращал строку, содержащую значение свойства `message`. Поэтому не следует ожидать, что метод `toString()` преобразует объект `Error` в строку, понятную человеку. Чтобы отобразить для пользователя сообщение об ошибке, необходимо явно использовать свойства `name` и `message` объекта `Error`.

Пример

Предупредить об исключении можно так:

```
function factorial(x) {
    if (x < 0) throw new Error(¨factorial: x должно быть >= 0¨);
    if (x <= 1) return 1; else return x * factorial(x1);
}
```

Перехватывая исключение, можно сообщить о нем пользователю следующим способом (с помощью клиентского метода `Window.alert()`):

```
try { &*(&/* здесь возникает ошибка */ }
catch(e) {
    if (e instanceof Error) { // Это экземпляр класса Error или его подкласса?
        alert(e.name + ¨: ¨ + e.message);
    }
}
```

См. также

`EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`

Error.message

сообщение об ошибке

Синтаксис

error.message

Описание

Свойство `message` объекта `Error` (или экземпляра любого подкласса `Error`) предназначено для хранения понятной человеку строки, содержащей подробные сведения о возникшей ошибке или исключении. Если конструктору `Error()` передан аргумент *message*, он становится значением свойства `message`. Если аргумент *message* передан не был, объект `Error` наследует для этого свойства значение по умолчанию, определенное реализацией (которое может быть пустой строкой).

Error.name

тип ошибки

Синтаксис

error.name

Описание

Свойство `name` объекта `Error` (или экземпляра любого подкласса `Error`) задает тип произошедшей ошибки или исключения. Все объекты `Error` наследуют это свойство от своего конструктора. Значение свойства совпадает с именем конструктора. Другими словами, у объектов `SyntaxError` свойство `name` равно «SyntaxError», а у объектов `EvalError` – «EvalError».

Error.toString()

преобразует объект Error в строку **переопределяет Object.toString()**

Синтаксис

error`.toString()`

Возвращаемое значение

Строка, определенная реализацией. Стандарт ECMAScript ничего не говорит о значении, возвращаемом этим методом, за исключением того, что оно должно быть строкой. Стоит отметить, что он не требует, чтобы возвращаемая строка содержала имя ошибки или сообщение об ошибке.

escape() устарело

кодирует строку

Синтаксис

`escape(`*s*`)`

Аргументы

s Строка, которая должна быть закодирована (с применением управляющих последовательностей).

Возвращаемое значение

Закодированная копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Описание

`escape()` – глобальная функция, которая возвращает новую строку, содержащую закодированную версию аргумента *s*. Сама строка *s* не изменяется. Функция `escape()` возвращает строку, в которой все символы, имеющиеся в строке *s*, отличные от букв, цифр и символов пунктуации (@, *, _, +, –, . и /) набора ASCII, заменены управляющими последовательностями в формате `%`*xx* или `%u`*xxxx* (где *x* обозначает шестнадцатеричную цифру). Символы Юникода от `\u0000` до `\u00ff` заменяются управляющей последовательностью `%`*xx*, все остальные символы Юникода – последовательностью `%u`*xxxx*.

Строка, закодированная с помощью `escape()`, декодируется функцией `unescape()`.

Хотя функция `escape()` стандартизована в первой версии ECMAScript, она была признана устаревшей и удалена из стандарта в ECMAScript v3. Реализации ECMAScript обычно поддерживают эту функцию, хотя это необязательно. Вместо `escape()` следует использовать функции `encodeURI()` и `encodeURIComponent()`.

Пример

```
escape("Hello World!"); // Вернет "Hello%20World%21"
```

См. также

`encodeURI()`, `encodeURIComponent()`

eval()

исполняет содержащийся в строке JavaScript-код

Синтаксис

`eval(`*`код`*`)`

Аргументы

код	Строка, содержащая выполняемое выражение или инструкции.

Возвращаемое значение

Значение, полученное в результате выполнения *кода*, если оно есть.

Исключения

Функция `eval()` возбуждает исключение `SyntaxError`, если аргумент *код* содержит некорректный программный код на языке JavaScript. Если исключение будет возбуждено самим программным кодом в аргументе *код*, функция `eval()` передаст его выше по стеку вызовов.

Описание

`eval()` – это метод глобального объекта, вычисляющий строку, в которой содержится программный код на языке JavaScript. Если *код* содержит JavaScript-выражение, `eval()` вычислит выражение и вернет его значение. Если *код* содержит одну или несколько JavaScript-инструкций, `eval()` выполнит эти инструкции и вернет то значение (если оно есть), которое вернет последняя инструкция. Если *код* не возвращает никакого значения, `eval()` вернет значение `undefined`. И наконец, если *код* сгенерирует исключение, `eval()` передаст это исключение вызывающей программе.

В спецификациях ECMAScript 3 и ECMAScript 5 определяется различное поведение функции `eval()`. Кроме того, в спецификации ECMAScript 5 для нее определяется различное поведение в строгом и нестрогом режимах. Чтобы объяснить эти различия, необходимо немного отклониться от темы. Реализовать эффективный интерпретатор намного проще, когда язык программирования определяет `eval` как оператор, а не как функцию. В языке JavaScript `eval` является функцией, но ради обеспечения эффективности в нем различаются *непосредственные* и *косвенные* вызовы `eval()`. В непосредственном вызове используется идентификатор `eval`, и, если убрать скобки, вызов функции будет выглядеть как применение оператора `eval`. Любые другие вызовы `eval()` являются косвенными. Если присвоить функцию `eval()` переменной с другим именем и вызвать по имени переменной, это будет косвенный вызов. Аналогично, если вызвать `eval()` как метод глобального объекта, это также будет косвенный вызов.

Определив понятия непосредственного и косвенного вызова, поведение функции `eval()` можно описать следующим образом:

Непосредственный вызов, ES3 и нестрогий режим ES5

`eval()` выполняет *код* в текущей лексической области видимости. Если *код* объявляет переменную или функцию, она будет определена в локальной области видимости. Это обычный случай использования функции `eval()`.

Косвенный вызов, ES3

Спецификация ECMAScript 3 позволяет интерпретаторам возбуждать исключение `EvalError` для любых косвенных вызовов `eval()`. На практике реализации ES3 обычно этого не делают, тем не менее в них следует избегать косвенных вызовов.

Косвенный вызов, ES5

Согласно спецификации ECMAScript 5, вместо того чтобы возбудить исключение `EvalError`, косвенный вызов `eval()` должен выполнить *код* в глобальной области видимости, игнорируя любые локальные переменные в текущей лексической области видимости. В ES5 можно выполнить присваивание `var geval = eval;` и затем использовать `geval()` для выполнения *кода* в глобальной области видимости.

Непосредственный и косвенный вызов, строгий режим

В строгом режиме объявления переменных и функций в *коде* определяют их в частной области видимости, которая действует только в данном вызове функции `eval()`. Это означает, что в строгом режиме непосредственный вызов `eval()` не может изменить лексическую область видимости, а косвенный вызов не может изменить глобальную область видимости. Эти правила действуют, если вызов `eval()` выполняется в строгом режиме или если *код* начинается с директивы «use strict».

Функция `eval()` в языке JavaScript предоставляет очень мощные возможности, тем не менее она не часто используется в реальных программах. Очевидной областью ее применения являются программы, работающие как рекурсивные интерпретаторы JavaScript или динамически генерирующие и выполняющие JavaScript-код.

Большинство JavaScript-функций, принимающих строковые аргументы, могут также принимать аргументы других типов и перед обработкой просто преобразуют эти значения в строки. Метод `eval()` ведет себя иначе. Если аргумент *код* не является элементарным строковым значением, он возвращается в неизменном виде. Поэтому будьте внимательны, чтобы случайно не передать функции `eval()` объект `String` вместо элементарного строкового значения.

Пример

```
eval("1+2"); // Вернет 3
// Этот фрагмент использует клиентские JavaScript-методы для запроса выражения
// от пользователя и вывода результатов его вычисления.
// Подробности см. в описаниях клиентских методов Window.alert() и Window.prompt().
try {
    alert("Результат: " + eval(prompt("Введите выражение:","")));
}
catch(exception) {
    alert(exception);
}
```

EvalError

генерируется при некорректном использовании eval() **Object→Error→EvalError**

Конструктор

`new EvalError()`

`new EvalError(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке с дополнительной информацией об исключении. Если этот аргумент указан, он принимается в качестве значения свойства `message` объекта `EvalError`.

Возвращаемое значение

Вновь созданный объект `EvalError`. Если задан аргумент *сообщение*, объект `Error` возьмет его в качестве значения своего свойства `message`; в противном случае в качестве значения этого свойства будет использована строка по умолчанию, определенная реализацией. Когда конструктор `EvalError()` вызывается как функция (без оператора `new`), он ведет себя точно так же, как при вызове с оператором `new`.

Свойства

`message`	Сообщение об ошибке с дополнительной информацией об исключении. В этом свойстве хранится строка, переданная конструктору, или строка по умолчанию, определенная реализацией. Подробности см. в статье с описанием свойства `Error.message`.
`name`	Строка, определяющая тип исключения. Для всех объектов `EvalError` значение этого свойства равно «EvalError».

Описание

Экземпляры класса `EvalError` могут создаваться, когда глобальная функция `eval()` вызывается с любым другим именем. Ограничения на способы вызова функции `eval()` рассматриваются в ее описании. Информация о генерации и перехвате исключений приводится в статье, посвященной классу `Error`.

См. также

`Error`, `Error.message`, `Error.name`

Function

функция JavaScript **Object→Function**

Синтаксис

```
function имя_функции(имена_аргументов) // Инструкция определения функции
{
   тело
}
function(имена _аргументов) { тело } // Литерал неименованной функции
имя_функции(значения_аргументов) // Вызов функции
```

Конструктор

```
new Function(имена_аргументов..., тело)
```

Аргументы

имена_аргументов...

Любое количество строковых аргументов, которые присваивают имя одному или нескольким аргументам создаваемого объекта `Function`.

тело

Строка, определяющая тело функции. Она может содержать любое количество инструкций на языке JavaScript, разделенных точками с запятой, и ссылаться на любые имена аргументов, указанные ранее в конструкторе.

Возвращаемое значение

Вновь созданный объект `Function`. Вызов функции приводит к выполнению JavaScript-кода, составляющего аргумент *тело*.

Исключения

`SyntaxError`	Указывает, что в аргументе *тело* или в одном из аргументов из перечня *имена_аргументов* имеется синтаксическая ошибка.

Свойства

`arguments[]`	Массив аргументов, переданных функции. Признано устаревшим.
`caller`	Ссылка на объект `Function`, вызвавший данную функцию, или `null`, если функция была вызвана из программного кода верхнего уровня. Признано устаревшим.
`length`	Число именованных аргументов, указанных при объявлении функции.
`prototype`	Объект, определяющий свойства и методы конструктора, совместно используемые всеми объектами, созданными с помощью этого конструктора.

Методы

`apply()`	Вызывает функцию как метод указанного объекта, передавая ей указанный массив аргументов.
`bind()`	Возвращает новую функцию, которая вызывает данную как метод указанного объекта с указанными аргументами.
`call()`	Вызывает функцию как метод указанного объекта, передавая ей указанные аргументы.
`toString()`	Возвращает строковое представление функции.

Описание

Функция в языке JavaScript – это фундаментальный тип данных. В главе 8 рассказывается, как определять и использовать функции, а в главе 9 рассматриваются близкие темы, касающиеся методов, конструкторов и свойства `prototype` функций. Подробности см. в этих главах. Обратите внимание: функциональные объекты могут создаваться с помощью описанного здесь конструктора `Function()`, но это неэффективно, поэтому в большинстве случаев предпочтительным способом определения функции является инструкция определения функции или функциональный литерал.

В JavaScript 1.1 и более поздних версиях тело функции автоматически получает локальную переменную по имени `arguments`, которая ссылается на объект `Arguments`. Этот объект представляет собой массив значений, переданных функции в качестве аргументов. Не путайте его с устаревшим свойством `arguments[]`, описанным ранее. Подробности см. в статье об объекте `Arguments`.

См. также

`Arguments`; главы 8 и 9

Function.apply()

вызывает функцию как метод объекта

Синтаксис

функция.`apply(`*этот_объект, аргументы*`)`

Аргументы

этот_объект	Объект, к которому должна быть применена функция. В теле функции аргумент *этот_объект* становится значением ключевого слова `this`. Если указанный аргумент содержит значение `null`, используется глобальный объект.
аргументы	Массив значений, которые должны передаваться *функции* в качестве аргументов.

Возвращаемое значение

Значение, возвращаемое при вызове *функции*.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося функцией, или с аргументом *аргументы*, не являющимся массивом или объектом `Arguments`.

Описание

Метод `apply()` вызывает указанную *функцию*, как если бы она была методом объекта, заданного аргументом *этот_объект*, передавая ей аргументы, которые содержатся в массиве *аргументы*. Он возвращает значение, возвращаемое *функцией*. В теле *функции* ключевое слово `this` ссылается на объект *этот_объект*.

Аргумент *аргументы* должен быть массивом или объектом `Arguments`. Если аргументы должны передаваться функции в виде отдельных аргументов, а не в виде массива, следует использовать вызов `Function.call()`.

Пример

```
// Применяет метод Object.toString(), предлагаемый по умолчанию для объекта,
// переопределяющего его собственной версией метода. Обратите внимание
// на отсутствие аргументов.
Object.prototype.toString.apply(o);
// Вызывает метод Math.max(), используемый для нахождения максимального элемента
// в массиве. Обратите внимание: в этом случае первый аргумент не имеет значения.
var data = [1,2,3,4,5,6,7,8];
Math.max.apply(null, data);
```

См. также

`Function.call()`

Function.arguments[] устарело

аргументы, переданные функции

Синтаксис

функция.`arguments[`*i*`]`

функция.`arguments.length`

Описание

Свойство `arguments` объекта `Function` представляет собой массив аргументов, переданных функции. Этот массив определен только во время выполнения функции. Свойство `arguments.length` позволяет определить количество элементов в массиве.

Это свойство признано устаревшим, и его никогда не следует использовать в новых JavaScript-сценариях; вместо него рекомендуется использовать объект `Arguments`.

См. также

`Arguments`

Function.bind() ECMAScript 5

возвращает функцию, которая вызывается как метод

Синтаксис

функция.`bind`(*объект*)

функция.`bind`(*объект*, *аргументы*...)

Аргументы

объект	Объект, к которому должна быть привязана *функция*.
аргументы...	Ноль или более значение аргументов, которые также должны быть связаны с *функцией*.

Возвращаемое значение

Новая функция, которая будет вызывать эту *функцию* как метод *объекта* и передавать ей указанные *аргументы*.

Описание

Метод `bind()` возвращает новую функцию, которая будет вызывать эту *функцию* как метод *объекта*. В качестве аргументов эта *функция* будет получать *аргументы*, переданные методу `bind()`, за которыми будут следовать аргументы, переданные новой функции.

Пример

Допустим, что имеется функция `f` и в программе вызывается ее метод `bind()`, как показано ниже:

```
var g = f.bind(o, 1, 2);
```

В результате этого будет создана новая функция `g`, вызов `g(3)` которой эквивалентен следующему вызову:

```
f.call(o, 1, 2, 3);
```

См. также

`Function.apply()`, `Function.call()`, раздел 8.7.4

Function.call()

вызывает функцию как метод объекта

Синтаксис

функция.`call`(*этот_объект*, *аргументы*...)

Аргументы

этот_объект	Объект, относительно которого должна быть вызвана *функция*. В теле функции аргумент *этот_объект* становится значением ключевого слова `this`. Если этот аргумент содержит значение `null`, используется глобальный объект.
аргументы...	Любое количество аргументов, передаваемых *функции*.

Возвращаемое значение

Значение, возвращаемое вызовом *функции*.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося функцией.

Описание

`call()` вызывает указанную *функцию*, как если бы она была методом объекта, указанного в аргументе *этот_объект*, передавая ей любые аргументы, расположенные в списке аргументов после аргумента *этот_объект*. Вызов `call()` возвращает то, что возвращает вызываемая функция. В теле функции ключевое слово `this` ссылается на объект *этот_объект* или на глобальный объект, если аргумент *этот_объект* содержит значение `null`.

Если аргументы для передачи в функцию требуется указать в виде массива, используйте метод `Function.apply()`.

Пример

```
// Вызывает метод Object.toString(), по умолчанию предлагаемый для объекта,
// переопределяющего его собственной версией метода. Обратите внимание
// на отсутствие аргументов.
Object.prototype.toString.call(o);
```

См. также

`Function.apply()`

Function.caller

устарело; не определено в строгом режиме

функция, вызвавшая данную

Синтаксис

функция.`caller`

Описание

В ранних версиях JavaScript свойство `caller` объекта `Function` представляло собой ссылку на функцию, вызвавшую текущую функцию. Если функция вызывается из JavaScript-программы верхнего уровня, свойство `caller` будет иметь значение `null`. Это свойство может использоваться только внутри функции (т. е. свойство `caller` определено для функции, только пока она выполняется).

Свойство `Function.caller` не является частью стандарта ECMAScript и не обязательно для совместимых реализаций, поэтому не следует использовать его.

Function.length

количество аргументов в объявлении функции

Синтаксис

функция.`length`

Описание

Свойство `length` функции указывает количество именованных аргументов, объявленных при определении функции. Фактически функция может вызываться с большим или меньшим количеством аргументов. Не путайте это свойство объекта `Function` со свойством `length` объекта `Arguments`, указывающим количество аргументов, фактически переданных функции. Пример имеется в статье о свойстве `Arguments.length`.

См. также

`Arguments.length`

Function.prototype

прототип класса объектов

Синтаксис

функция.`prototype`

Описание

Свойство `prototype` применяется, когда функция вызывается как конструктор. Оно ссылается на объект, являющийся прототипом для целого класса объектов. Любой объект, созданный с помощью конструктора, наследует все свойства объекта, на который ссылается свойство `prototype`.

Обсуждение функций-конструкторов, свойства `prototype` и определений классов в языке JavaScript находится в главе 9.

См. также

Глава 9

Function.toString()

преобразует функцию в строку

Синтаксис

функция.`toString()`

Возвращаемое значение

Строка, представляющая функцию.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося функцией.

Описание

Метод `toString()` объекта `Function` преобразует функцию в строку способом, зависящим от реализации. В большинстве реализаций, например в Firefox и IE, данный метод возвращает строку JavaScript-кода, которая включает ключевое слово `function`, список аргументов, полное тело функции и т. д. В этих реализациях результат работы метода `toString()` может передаваться в виде аргумента функции `eval()`. Однако такое поведение не оговаривается спецификациями, и на него не следует полагаться.

Global

глобальный объект **Object→Global**

Синтаксис

```
this
```

Глобальные свойства

Глобальный объект – это не класс, поэтому для следующих глобальных свойств имеются отдельные справочные статьи под собственными именами. То есть подробные сведения о свойстве `undefined` можно найти под заголовком «undefined», а не «Global. undefined». Обратите внимание, что все переменные верхнего уровня также представляют собой свойства глобального объекта.

`Infinity`	Числовое значение, обозначающее положительную бесконечность.
`NaN`	Нечисловое значение.
`undefined`	Значение `undefined`.

Глобальные функции

Глобальный объект – это объект, а не класс, поэтому перечисленные далее глобальные функции не являются методами какого-либо объекта и справочные статьи приведены под именами функций. Так, функция `parseInt()` подробно описывается в статье под заголовком «parseInt()», а не «Global.parseInt()».

`decodeURI()`	Декодирует строку, закодированную с помощью функции `encodeURI()`.
`decodeURIComponent()`	Декодирует строку, закодированную с помощью функции `encodeURIComponent()`.
`encodeURI`	Кодирует URI, заменяя определенные символы управляющими последовательностями.
`encodeURIComponent`	Кодирует компонент URI, заменяя определенные символы управляющими последовательностями.
`escape()`	Кодирует строку, заменяя определенные символы управляющими последовательностями.
`eval()`	Вычисляет строку с программным кодом на языке JavaScript и возвращает результат.
`isFinite()`	Проверяет, является ли значение конечным числом.
`isNaN`	Проверяет, является ли значение нечисловым (`NaN`).
`parseFloat()`	Выбирает число из строки.
`parseInt()`	Выбирает целое из строки.
`unescape()`	Декодирует строку, закодированную вызовом `escape()`.

Глобальные объекты

В дополнение к перечисленным ранее глобальным свойствам и функциям, глобальный объект определяет свойства, ссылающиеся на все остальные предопределенные

JavaScript-объекты. Большинство из этих свойств являются функциями-конструкторами:

`Array`	Конструктор `Array()`.
`Boolean`	Конструктор `Boolean()`.
`Date`	Конструктор `Date()`.
`Error`	Конструктор `Error()`.
`EvalError`	Конструктор `EvalError()`.
`Function`	Конструктор `Function()`.
`JSON`	Ссылка на объект, определяющий функции для сериализации объектов в формат JSON и обратно.
`Math`	Ссылка на объект, определяющий математические функции.
`Number`	Конструктор `Number()`.
`Object`	Конструктор `Object()`.
`RangeError`	Конструктор `RangeError()`.
`ReferenceError`	Конструктор `ReferenceError()`.
`RegExp`	Конструктор `RegExp()`.
`String`	Конструктор `String()`.
`SyntaxError`	Конструктор `SyntaxError()`.
`TypeError`	Конструктор `TypeError()`.
`URIError`	Конструктор `URIError()`.

Описание

Глобальный объект – это предопределенный объект, который в языке JavaScript служит для размещения глобальных свойств и функций. Все остальные предопределенные объекты, функции и свойства доступны через глобальный объект. Глобальный объект не является свойством какого-либо другого объекта, поэтому у него нет имени. (Заголовок справочной статьи выбран просто для удобства и не указывает на то, что глобальный объект имеет имя «Global».) В JavaScript-коде верхнего уровня можно ссылаться на глобальный объект посредством ключевого слова `this`. Однако этот способ обращения к глобальному объекту требуется редко, т. к. глобальный объект выступает в качестве начала цепочки областей видимости, поэтому поиск неуточненных имен переменных и функций выполняется среди свойств этого объекта. Когда JavaScript-код ссылается, например, на функцию `parseInt()`, он ссылается на свойство `parseInt` глобального объекта. Тот факт, что глобальный объект находится в начале цепочки областей видимости, означает также, что все переменные, объявленные в JavaScript-коде верхнего уровня, становятся свойствами глобального объекта.

Глобальный объект – это просто объект, а не класс. У него нет конструктора `Global()` и нет способа создать новый экземпляр глобального объекта.

Когда JavaScript-код встраивается в определенную среду, глобальному объекту обычно придаются дополнительные свойства, специфические для этой среды. На самом деле тип глобального объекта в стандарте ECMAScript не указан, и в конкретной реализации JavaScript в качестве глобального может выступать объект любого типа, если этот объект определяет перечисленные здесь основные свойства и функции. В клиентском JavaScript, например, глобальным объектом является объект `Window`, представляющий окно веб-броузера, внутри которого выполняется JavaScript-код.

Пример

В базовом JavaScript ни одно из предопределенных свойств глобального объекта не является перечислимым, поэтому можно получить список всех явно и неявно объявленных глобальных переменных с помощью следующего цикла `for/in`:

```
var variables = ""
for(var name in this)
    variables += name + "\n";
```

См. также

`Window` (часть IV книги); глава 3

Infinity

числовое свойство, обозначающее бесконечность

Синтаксис

```
Infinity
```

Описание

`Infinity` – это глобальное свойство, содержащее специальное числовое значение, которое обозначает положительную бесконечность. Свойство `Infinity` не перечисляется циклами `for/in` и не может быть удалено с помощью оператора `delete`. Следует отметить, что `Infinity` не является константой и может быть установлено равным какому-либо другому значению, но лучше этого не делать. (В то же время `Number.POSITIVE_INFINITY` – это константа.)

См. также

`isFinite()`, `NaN`, `Number.POSITIVE_INFINITY`

isFinite()

определяет, является ли число конечным

Синтаксис

`isFinite(`*n*`)`

Аргументы

n Проверяемое число.

Возвращаемое значение

Если *n* является конечным числом (или может быть преобразовано в него), возвращает `true`, если *n* не является числом (`NaN`) или плюс/минус бесконечностью – `false`.

См. также

`Infinity`, `isNaN()`, `NaN`, `Number.NaN`, `Number.NEGATIVE_INFINITY`, `Number.POSITIVE_INFINITY`

isNaN()

определяет, является ли аргумент нечисловым значением

Синтаксис

`isNaN(`*x*`)`

Аргументы

x Проверяемое значение.

Возвращаемое значение

Если *x* является специальным нечисловым значением (или может быть в него преобразовано), возвращает `true`, если *x* является любым другим значением – `false`.

Описание

Название «NaN» является аббреквиатурой от «Not-a-Number» (не число). Глобальная переменная `NaN` хранит специальное числовое значение (которое также называется `NaN`), представляющее недопустимое число (например, результат деления на ноль). `isNaN()` проверяет свой аргумент, чтобы определить, является ли он нечисловым. Эта функция возвращает `false`, если аргумент *x* является или может быть преобразован в числовое значение, отличное от `NaN`. Она возвращает `true`, если аргумент *x* не является или не может быть преобразован в числовое значение или если он равен `NaN`.

Важной особенностью `NaN` является то обстоятельство, что это значение не равно никакому значению, даже самому себе. Поэтому, если потребуется проверить некоторое значение на равенство `NaN`, нельзя будет использовать привычную проверку `x === NaN`: она всегда будет возвращать `false`. Вместо этого следует использовать выражение `x !== x`: оно возвращает `true`, только когда `x` равно `NaN`.

Обычно функция `isNaN()` служит для проверки результатов, возвращаемых функциями `parseFloat()` и `parseInt()`, с целью определить, представляют ли эти результаты корректные числа.

Пример

```
isNaN(0);                  // => false
isNaN(0/0);                // => true
isNaN(parseInt("3"));      // => false
isNaN(parseInt("hello"));  // => true
isNaN("3");                // => false
isNaN("hello");            // => true
isNaN(true);               // => false
isNaN(undefined);          // => true
```

См. также

`isFinite()`, `NaN`, `Number.NaN`, `parseFloat()`, `parseInt()`

JSON **ECMAScript 5**

выполняет преобразование в формат JSON и обратно

Описание

`JSON` – это простой объект, который играет роль пространства имен для глобальных функций `JSON.parse()` и `JSON.stringify()`, определяемых стандартом ECMAScript 5. Свойство `JSON` не является конструктором. Ранее, до появления стандарта ECMAScript 5, совместимые функции преобразования в формат JSON и обратно были доступны в библиотеке *http://json.org/json2.js*.

Аббревиатура «JSON» происходит от JavaScript Object Notation (форма записи JavaScript-объектов). JSON – это формат сериализации данных, основанный на синтаксисе литералов в языке JavaScript, который может представлять значение `null`, логические значения `true` и `false`, вещественные числа (с использованием формы записи числовых литералов в языке JavaScript), строки (с использованием формы записи строковых литералов), массивы значений (с использованием формы записи литералов массивов) и отображения строк в значения (с использованием формы записи литералов объектов). Элементарное значение `undefined`, а также числовые значения `NaN` и `Infinity` не могут быть представлены в формате JSON. Функции, объекты `Date`, `RegExp` и `Error` также не поддерживаются.

Пример

```
// Создает полную копию любого объект или массива, который может быть
// представлен в формате JSON
function deepcopy(o) { return JSON.parse(JSON.stringify(o)); }
```

См. также

`JSON.parse()`, `JSON.stringify()`, раздел 6.9, ***http://json.org***

JSON.parse() ECMAScript 5

выполняет анализ строки в формате JSON

Синтаксис

JSON.parse(*s*)

JSON.parse(*s*, *reviver*)

Аргументы

s	Анализируемая строка.
reviver	Необязательная функция, способная преобразовывать значения, полученные в ходе анализа.

Возвращаемое значение

Объект, массив или элементарное значение, полученное в результате анализа строки *s* (и, возможно, измененное функцией *reviver*).

Описание

`JSON.parse()` – глобальная функция, предназначенная для анализа строк в формате JSON. Обычно ей передается единственный строковый аргумент, и она возвращает значение, представленное строкой.

Необязательный аргумент *reviver* можно использовать для фильтрации или заключительной обработки значения перед тем, как оно будет возвращено вызывающей программе. Если этот аргумент указан, функция *reviver* будет вызвана для каждого элементарного значения (но не для объектов или массивов, содержащих эти элементарные значения), полученного в ходе анализа строки *s*. При вызове функции *reviver* будет передано два аргумента. Первый – имя свойства, т. е. имя свойства объекта или индекс массива в виде строки. Второй – элементарное значение этого свойства объекта или элемента массива. Кроме того, функция *reviver* будет вызываться как метод объекта или массива, содержащего элементарное значение. Особый случай, когда строка *s* является представлением элементарного значения, а не объекта или массива. В этом случае элементарное значение будет сохранено во вновь созданном объекте в свойстве, именем которого является пустая строка, и функция *reviver* будет вызвана для относительно этого вновь созданного объекта с пустой строкой в первом аргументе и элементарным значением – во втором.

Возвращаемое значение функции *reviver* станет новым значением заданного свойства. Если функция *reviver* вернет значение своего второго аргумента, то значение свойства не изменится. Если функция *reviver* вернет значение `undefined` (или вообще не вернет никакого значения), то данное свойство будет удалено из объекта или массива перед тем, как `JSON.parse()` вернет его вызывающей программе.

Пример

Во многих случаях использование `JSON.parse()` выглядит достаточно просто:

```
var data = JSON.parse(text);
```

Функция `JSON.stringify()` преобразует объекты `Date` в строки, а вы с помощью собственной функции *reviver* можете выполнять обратное преобразование. Пример ниже также фильтрует свойства по их именам и возвращает `undefined`, чтобы удалить определенные свойства из полученного объекта:

```
var data JSON.parse(text, function(name, value) {
    // Удалить свойства, имена которых начинаются с символа подчеркивания
    if (name[0] == '_') return undefined;
    // Если значение является строкой в формате представления дат ISO 8601,
    // преобразовать его в объект Date.
    if (typeof value === "string" &&
        /^\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\dZ$/.test(value))
        return new Date(value);
    // Иначе вернуть неизмененное значение
    return value
});
```

См. также

`JSON.stringify()`, **раздел 6.9**

JSON.stringify() ECMAScript 5

сериализует объект, массив или элементарное значение

Синтаксис

`JSON.stringify(`*o*`)`
`JSON.stringify(`*o*`, `*filter*`)`
`JSON.stringify(`*o*`, `*filter*`, `*indent*`)`

Arguments

o	**Объект, массив или элементарное значение, которое требуется преобразовать в строку в формате JSON.**
filter	**Необязательная функция, которая может изменять значения перед сериализацией, или массив имен свойств, подлежащих сериализации.**
indent	**Необязательный аргумент, определяющий строку или число пробелов для оформления отступов, когда требуется применить форматирование для удобочитаемости. Если отсутствует, возвращаемая строка не содержит дополнительных пробелов, но вполне понятна для человека.**

Возвращаемое значение

Строка в формате JSON, представляющая значение *o* после применения фильтра *filter* и отформатированная с применением *indent*.

Описание

Функция `JSON.stringify()` преобразует элементарное значение, объект или массив в строку в формате JSON, которая позднее сможет быть преобразована обратно в значение с помощью функции `JSON.parse()`. Обычно эта функция вызывается с единственным аргументом и возвращает соответствующую строку.

Когда функция `JSON.stringify()` вызывается с единственным аргументом и когда в нем передается объект, массив, строка, число, логическое значение или значение `null`, сериализация выполняется очень просто. Однако, когда значение для сериализации содержит объекты, являющиеся экземплярами класса, процесс сериализаци усложняется. Если функция `JSON.stringify()` встретит объект (или массив) с методом `toJSON()`, она вызовет этот метод и выполнит сериализацию полученного в результате значения, а не самого объекта. Она вызывает метод `toJSON()` с единственным строковым аргументом, в котором передается имя свойства объекта или индекс массива. Класс `Date` определяет метод `toJSON()`, преобразующий объекты `Date` в строки с помощью метода `Date.toISOString()`. Никакие другие классы, встроенные в язык JavaScript, не определяют метод `toJSON()`, но вы можете определить его в своих классах. Помните, что несмотря на свое имя, метод `toJSON()` не выполняет сериализацию объекта: он просто возвращает значение, которое будет подвергнуто сериализации вместо оригинального объекта.

Второй аргумент функции `JSON.stringify()` позволяет добавить в процесс сериализаци второй слой фильтрации. Этот необязательный аргумент может быть функцией или массивом и предоставляет возможность реализации двух разных способов фильтрации. Если передать во втором аргументе функцию, она будет использоваться как инструмент замены, подобно методу `toJSON()`, описанному выше. Эта функция будет вызываться для каждого значения, подлежащего сериализации. Ключевое слово `this` внутри этой функции замены будет ссылаться на объект или массив, в котором определено текущее значение. Первым аргументом функции замены будет передаваться имя свойства объекта или индекс в массиве для текущего значения, а во втором – само значение. Это значение будет замещено возвращаемым значением функции. Если функция вернет `undefined` или вообще ничего не вернет, то это значение (и соответствующее ему свойство объекта или элемент массива) будет исключено из сериализации.

Если во втором аргументе передать массив строк (или чисел – они будут преобразованы в строки), эти строки будут использоваться как имена свойств объекта. Любое свойство, имя которого отсутствует в массиве, будет исключено из сериализации. Кроме того, возвращаемая строка будет содержать свойства в том же порядке, в каком они перечислены в массиве.

Функция `JSON.stringify()` обычно возвращает строку без дополнительных пробелов и символов перевода строки. Если желательно сделать результат более удобочитаемым, можно передать третий агумент. Если указать в нем число от 1 до 10, функция `JSON.stringify()` вставит символы перевода строки и будет использовать указанное число пробелов для оформления отступов на каждом «уровне» вложенности. Если в этом аргументе передать непустую строку, функция `JSON.stringify()` вставит символы перевода строки и будет использовать указанную строку (но не более 10 первых символов из нее) для оформления отступов.

Примеры

```
// Простая сериализация
var text = JSON.stringify(data);

// Указать точно, какие поля подлежат сериализации
var text = JSON.stringify(address, ["city","state","country"]);
```

```
// Указать функцию замены, чтобы можно было сериализовать объекты RegExp
var text = JSON.stringify(patterns, function(key, value) {
    if (value.constructor === RegExp) return value.toString();
    return value;
});

// Того же эффекта можно добиться иначе:
RegExp.prototype.toJSON = function() { return this.toString(); }
```

См. также

`JSON.parse()`, раздел 6.9

Math

математические функции и константы

Синтаксис

`Math.`*`константа`*

`Math.`*`функция`*`()`

Константы

`Math.E`	Константа *e*, основание натуральных логарифмов.
`Math.LN10`	Натуральный логарифм числа 10.
`Math.LN2`	Натуральный логарифм числа 2.
`Math.LOG10E`	Десятичный логарифм числа *e*.
`Math.LOG2E`	Логарифм числа *e* по основанию 2.
`Math.PI`	Константа π.
`Math.SQRT1_2`	Единица, деленная на корень квадратный из 2.
`Math.SQRT2`	Квадратный корень из 2.

Статические функции

`Math.abs()`	Вычисляет абсолютное значение.
`Math.acos()`	Вычисляет арккосинус.
`Math.asin()`	Вычисляет арксинус.
`Math.atan()`	Вычисляет арктангенс.
`Math.atan2()`	Вычисляет угол между осью X и точкой.
`Math.ceil()`	Округляет число вверх.
`Math.cos()`	Вычисляет косинус.
`Math.exp()`	Вычисляет степень числа *e*.
`Math.floor()`	Округляет число вниз.
`Math.log()`	Вычисляет натуральный логарифм.
`Math.max()`	Возвращает большее из двух чисел.
`Math.min()`	Возвращает меньшее из двух чисел.
`Math.pow()`	Вычисляет *x* в степени *y*.
`Math.random()`	Возвращает случайное число.
`Math.round()`	Округляет до ближайшего целого.
`Math.sin()`	Вычисляет синус.

`Math.sqrt()`	Вычисляет квадратный корень.
`Math.tan()`	Вычисляет тангенс.

Описание

`Math` – это объект, определяющий свойства, которые ссылаются на математические функции и константы. Эти функции и константы вызываются с помощью следующего синтаксиса:

```
y = Math.sin(x);
area = radius * radius * Math.PI;
```

`Math` – это не класс объектов, как `Date` и `String`. Объект `Math` не имеет конструктора `Math()`, поэтому такие функции, как `Math.sin()`, – это просто функции, а не методы объекта.

См. также

`Number`

Math.abs()

вычисляет абсолютное значение

Синтаксис

`Math.abs(`*x*`)`

Аргументы

x	Любое число.

Возвращаемое значение

Абсолютное значение *x*.

Math.acos()

вычисляет арккосинус

Синтаксис

`Math.acos(`*x*`)`

Аргументы

x	Число от -1,0 до 1,0.

Возвращаемое значение

Арккосинус указанного числа *x*. Возвращаемое значение может находиться в интервале от 0 до π радиан.

Math.asin()

вычисляет арксинус

Синтаксис

`Math.asin(`*x*`)`

Аргументы

x	Число от -1,0 до 1,0.

Возвращаемое значение

Арксинус указанного значения *x*. Это возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan()

вычисляет арктангенс

Синтаксис

`Math.atan(x)`

Аргументы

x	Любое число.

Возвращаемое значение

Арктангенс указанного значения *x*. Возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan2()

вычисляет угол между осью X и точкой

Синтаксис

`Math.atan2(y, x)`

Аргументы

y	Координата Y точки.
x	Координата X точки.

Возвращаемое значение

Значение, лежащее между $-\pi$ и π радиан и указывающее на угол по направлению, обратному часовой стрелке, между положительной осью X и точкой (x,y).

Описание

Функция `Math.atan2()` вычисляет арктангенс отношения *y*/*x*. Аргумент *y* может рассматриваться как координата Y (или «рост») точки, а аргумент *x* – как координата X (или «пробег») точки. Обратите внимание на необычный порядок следования аргументов этой функции: координата Y передается до координаты X.

Math.ceil()

округляет число вверх

Синтаксис

`Math.ceil(x)`

Аргументы

x	Числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, большее или равное *x*.

Описание

Функция `Math.ceil()` вычисляет наименьшее целое, т. е. возвращает ближайшее целое, большее или равное аргументу функции. Функция `Math.ceil()` отличается от `Math.round()` тем, что округляет всегда вверх, а не к ближайшему целому. Обратите внимание также, что `Math.ceil()` округляет отрицательные числа не к большим по абсолютному значению отрицательным целым; функция округляет их по направлению к нулю.

Пример

```
a = Math.ceil(1.99);  // Результат равен 2.0
b = Math.ceil(1.01);  // Результат равен 2.0
c = Math.ceil(1.0);   // Результат равен 1.0
d = Math.ceil(-1.99); // Результат равен -1.0
```

Math.cos()

вычисляет косинус

Синтаксис

`Math.cos(`*x*`)`

Аргументы

x Угол в радианах. Чтобы преобразовать градусы в радианы, нужно умножить значение в градусах на 0,017453293 (2π/360).

Возвращаемое значение

Косинус указанного значения *x*. Это возвращаемое значение может находиться в интервале от -1,0 до 1,0.

Math.E

математическая константа e

Синтаксис

`Math.E`

Описание

`Math.E` – это математическая константа *e*, база натуральных логарифмов, приблизительно равная 2,71828.

Math.exp()

вычисляет e^x

Синтаксис

`Math.exp(`*x*`)`

Аргументы

x Число или выражение, которое должно использоваться как экспонента.

Возвращаемое значение

e^x – это число *e*, возведенное в степень указанной экспоненты *x*, где *e* – это основание натуральных логарифмов, примерно равное 2,71828.

Math.floor()

округляет число вниз

Синтаксис

`Math.floor(x)`

Аргументы

x Числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, меньшее или равное *x*.

Описание

Округление вниз, т. е. функция возвращает ближайшее целое значение, меньшее или равное аргументу функции.

Функция `Math.floor()` округляет вещественное число вниз, в отличие от функции `Math.round()`, выполняющей округление до ближайшего целого. Обратите внимание: `Math.floor()` округляет отрицательные числа вниз (т. е. дальше от нуля), а не вверх (т. е. ближе к нулю).

Пример

```
a = Math.floor(1.99);  // Результат равен 1.0
b = Math.floor(1.01);  // Результат равен 1.0
c = Math.floor(1.0);   // Результат равен 1.0
d = Math.floor(-1.01); // Результат равен -2.0
```

Math.LN10

математическая константа $\log_e 10$

Синтаксис

`Math.LN10`

Описание

`Math.LN10` – это $\log_e 10$, натуральный логарифм числа 10. Эта константа имеет значение, приблизительно равное 2,3025850929940459011.

Math.LN2

математическая константа $\log_e 2$

Синтаксис

`Math.LN2`

Описание

`Math.LN2` – это $\log_e 2$, натуральный логарифм числа 2. Эта константа имеет значение, приблизительно равное 0,69314718055994528623.

Math.log()

вычисляет натуральный логарифм

Синтаксис

`Math.log(x)`

Аргументы

x	Любое числовое значение, большее нуля.

Возвращаемое значение

Натуральный логарифм *x*.

Описание

`Math.log()` вычисляет натуральный логарифм своего аргумента. Аргумент должен быть больше нуля.

Логарифмы числа по основанию 10 и 2 можно вычислить по следующим формулам:

$$\log_{10}x = \log_{10}e \cdot \log_e x$$
$$\log_2 x = \log_2 e \cdot \log_e x$$

Эти формулы транслируются в следующие JavaScript-функции:

```
function log10(x) { return Math.LOG10E * Math.log(x); }
function log2(x) { return Math.LOG2E * Math.log(x); }
```

Math.LOG10E

математическая константа $\log_{10}e$

Синтаксис

`Math.LOG10E`

Описание

`Math.LOG10E` – это $\log_{10}e$, логарифм по основанию 10 константы *e*. Его значение приблизительно равно 0,43429448190325181667.

Math.LOG2E

математическая константа $\log_2 e$

Синтаксис

`Math.LOG2E`

Описание

`Math.LOG2E` – это $\log_2 e$, логарифм по основанию 2 константы *e*. Его значение приблизительно равно 1,442695040888963387.

Math.max()

возвращает наибольший аргумент

Синтаксис

`Math.max(аргументы...)`

Аргументы

аргументы...	Ноль или более значений.

Возвращаемое значение

Наибольший из аргументов. Возвращает `-Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов равен `NaN` или является нечисловым значением, которое не может быть преобразовано в число.

Math.min()

возвращает наименьший аргумент

Синтаксис

`Math.min(аргументы...)`

Аргументы

аргументы...	Любое количество аргументов.

Возвращаемое значение

Наименьший из указанных аргументов. Возвращает `Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов представляет собой значение `NaN` или нечисловое значение, которое не может быть преобразовано в число.

Math.PI

математическая константа π

Синтаксис

`Math.PI`

Описание

`Math.PI` – это константа π, т. е. отношение длины окружности к ее диаметру. Имеет значение, примерно равное 3,14159265358979.

Math.pow()

вычисляет x^y

Синтаксис

`Math.pow(x, y)`

Аргументы

x	Число, которое должно быть возведено в степень.
y	Степень, в которую должно быть возведено число *x*.

Возвращаемое значение

x в степени *y* (x^y).

Описание

`Math.pow()` вычисляет *x* в степени *y*. Значения *x* и *y* могут быть любыми. Однако если результат является мнимым или комплексным числом, `Math.pow()` возвращает `NaN`. На практике это означает, что если значение *x* отрицательно, то значение *y* должно быть положительным или отрицательным целым. Также имейте в виду, что большие экспоненты легко приводят к вещественному переполнению и возвращают значение `Infinity`.

Math.random()

возвращает псевдослучайное число

Синтаксис

`Math.random()`

Возвращаемое значение

Псевдослучайное число от 0,0 до 1,0.

Math.round()

округляет число до ближайшего целого

Синтаксис

`Math.round(`*x*`)`

Аргументы

x	Любое число.

Возвращаемое значение

Целое, ближайшее к *x*.

Описание

`Math.round()` округляет аргумент вверх или вниз до ближайшего целого. Число 0,5 округляется вверх. Например, число 2,5 округляется до 3, а число -2,5 округляется до -2.

Math.sin()

вычисляет синус

Синтаксис

`Math.sin(`*x*`)`

Аргументы

x	Угол в радианах. Для преобразования градусов в радианы умножьте число на 0,017453293 ($2\pi/360$).

Возвращаемое значение

Синус *x* – число в диапазоне от -1,0 до 1,0.

Math.sqrt()

вычисляет квадратный корень

Синтаксис

`Math.sqrt(`*x*`)`

Аргументы

x	Числовое значение, большее или равное 0.

Возвращаемое значение

Квадратный корень из *x*. Возвращает `NaN`, если *x* меньше нуля.

Описание

`Math.sqrt()` вычисляет квадратный корень числа. Следует заметить, что произвольные корни чисел можно вычислять посредством функции `Math.pow()`. Например:

```
Math.cuberoot = function(x){ return Math.pow(x,1/3); }
Math.cuberoot(8); // Вернет 2
```

Math.SQRT1_2

математическая константа 1/√2

Синтаксис

`Math.SQRT1_2`

Описание

`Math.SQRT1_2` – это 1/√2, величина, обратная корню квадратному из 2. Эта константа примерно равна 0,7071067811865476.

Math.SQRT2

математическая константа √2

Синтаксис

`Math.SQRT2`

Описание

`Math.SQRT2` – это √2, корень квадратный из 2. Эта константа имеет значение, примерно равное 1,414213562373095.

Math.tan()

вычисляет тангенс

Синтаксис

`Math.tan(x)`

Аргументы

x Угол, измеряемый в радианах. Чтобы преобразовать градусы в радианы, нужно умножить значение в градусах на 0,017453293 (2π/360).

Возвращаемое значение

Тангенс указанного угла *x*.

NaN

свойство «нечисло»

Синтаксис

`NaN`

Описание

`NaN` – это глобальное свойство, ссылающееся на специальное числовое значение «нечисло». Свойство `NaN` не перечисляется циклами `for/in` и не может быть удалено оператором `delete`. Обратите внимание: `NaN` – это не константа, и оно может быть установлено в любое значение, но лучше этого не делать.

Определить, является ли значение нечислом, можно с помощью функции `isNaN()`, т. к. `NaN` всегда при сравнении оказывается неравным любой другой величине, включая само себя!

См. также

`Infinity`, `isNaN()`, `Number.NaN`

Number

поддержка чисел **Object→Number**

Конструктор

`new Number(`*значение*`)`

`Number(`*значение*`)`

Аргументы

значение	Числовое значение создаваемого объекта `Number` или значение, которое может быть преобразовано в число.

Возвращаемое значение

Когда функция `Number()` используется в качестве конструктора (с оператором `new`), она возвращает вновь созданный объект `Number`. Когда функция `Number()` вызывается как функция (без оператора `new`), она преобразует свой аргумент в элементарное числовое значение и возвращает это значение (или `NaN`, если преобразование не удалось).

Константы

`Number.MAX_VALUE`

Наибольшее представимое число.

`Number.MIN_VALUE`

Наименьшее представимое число.

`Number.NaN`

Нечисло.

`Number.NEGATIVE_INFINITY`

Отрицательная бесконечность, возвращается при переполнении.

`Number.POSITIVE_INFINITY`

Положительная бесконечность; возвращается при переполнении.

Методы

`toString()`

Преобразует число в строку в указанной системе счисления.

`toLocaleString()`

Преобразует число в строку, руководствуясь локальными соглашениями о форматировании чисел.

`toFixed()`

Преобразует число в строку, содержащую указанное число цифр после десятичной точки. •

`toExponential()`

Преобразует числа в строки в экспоненциальной нотации с указанным количеством цифр после десятичной точки.

`toPrecision()`

Преобразует число в строку, записывая в нее указанное количество значащих цифр. Используется нотация экспоненциальная или с фиксированной точкой в зависимости от размера числа и заданного количества значащих цифр.

`valueOf()`

Возвращает элементарное числовое значение объекта `Number`.

Описание

Числа – это базовый элементарный тип данных в JavaScript. В JavaScript поддерживается также объект `Number`, представляющий собой обертку вокруг элементарного числового значения. Интерпретатор JavaScript при необходимости автоматически выполняет преобразование между элементарной и объектной формами. Существует возможность явно создать объект `Number` посредством конструктора `Number()`, хотя в этом редко возникает необходимость.

Конструктор `Number()` может также вызываться как функция преобразования (без оператора `new`). В этом случае функция пытается преобразовать свой аргумент в число и возвращает элементарное числовое значение (или `NaN`), полученное при преобразовании.

Конструктор `Number()` используется также для размещения пяти полезных числовых констант: максимального и минимального представимых чисел, положительной и отрицательной бесконечности, а также специального значения «нечисло». Обратите внимание: эти значения представляют собой свойства самой функции-конструктора `Number()`, а не отдельных числовых объектов. Например, свойство `MAX_VALUE` можно использовать следующим образом:

```
var biggest = Number.MAX_VALUE
```

А такая запись неверна:

```
var n = new Number(2);
var biggest = n.MAX_VALUE
```

В то же время `toString()` и другие методы объекта `Number` являются методами каждого объекта `Number`, а не функции-конструктора `Number()`. Как уже говорилось, JavaScript при необходимости автоматически выполняет преобразования между элементарными числовыми значениями и объектами `Number`. То есть методы класса `Number` могут работать с элементарными числовыми значениями так же, как с объектами `Number`:

```
var value = 1234;
var binary_value = n.toString(2);
```

См. также

`Infinity`, `Math`, `NaN`

Number.MAX_VALUE

максимальное числовое значение

Синтаксис

`Number.MAX_VALUE`

Описание

`Number.MAX_VALUE` – это наибольшее число, представимое в JavaScript. Его значение примерно равно 1,79E+308.

Number.MIN_VALUE

минимальное числовое значение

Синтаксис

`Number.MIN_VALUE`

Описание

`Number.MIN_VALUE` – это наименьшее число (ближайшее к нулю, а не самое отрицательное), представимое в JavaScript. Его значение примерно равно 5E–324.

Number.NaN

специальное нечисловое значение

Синтаксис

`Number.NaN`

Описание

`Number.NaN` – это специальное значение, указывающее, что результат некоторой математической операции (например, извлечения квадратного корня из отрицательного числа) не является числом. Функции `parseInt()` и `parseFloat()` возвращают это значение, когда не могут преобразовать указанную строку в число; программист может использовать `Number.NaN` аналогичным образом, чтобы указать на ошибочное условие для какой-либо функции, обычно возвращающей допустимое число.

JavaScript выводит значение `Number.NaN` как `NaN`. Обратите внимание: при сравнении значение `NaN` всегда не равно любому другому числу, включая само значение `NaN`. Следовательно, невозможно проверить значение на «нечисло», сравнив его с `Number.NaN`. Для этого предназначена функция `isNaN()`. В стандарте ECMAScript v1 и более поздних версиях вместо `Number.NaN` допускается использование предопределенного глобального свойства `NaN`.

См. также

`isNaN()`, `NaN`

Number.NEGATIVE_INFINITY

отрицательная бесконечность

Синтаксис

`Number.NEGATIVE_INFINITY`

Описание

`Number.NEGATIVE_INFINITY` – специальное числовое значение, возвращаемое, если арифметическая операция или математическая функция генерирует отрицательное число, большее чем максимальное представимое в JavaScript число (т. е. отрицательное число, меньшее чем `-Number.MAX_VALUE`).

JavaScript выводит значение `NEGATIVE_INFINITY` как `-Infinity`. Это значение математически ведет себя как бесконечность. Например, все что угодно, умноженное на бесконечность, является бесконечностью, а все, деленное на бесконечность, – нулем. В ECMAScript v1 и более поздних версиях можно также использовать предопределенную глобальную константу `-Infinity` вместо `Number.NEGATIVE_INFINITY`.

См. также

`Infinity`, `isFinite()`

Number.POSITIVE_INFINITY

бесконечность

Синтаксис

`Number.POSITIVE_INFINITY`

Описание

`Number.POSITIVE_INFINITY` – это специальное числовое значение, возвращаемое, когда арифметическая операция или математическая функция приводит к переполнению или генерирует значение, превосходящее максимальное представимое в JavaScript число (т. е. `Number.MAX_VALUE`). Обратите внимание: если происходит потеря значимости или число становится меньше, чем `Number.MIN_VALUE`, JavaScript преобразует его в ноль.

JavaScript выводит значение `POSITIVE_INFINITY` как `Infinity`. Это значение ведет себя математически так же, как бесконечность. Например, что-либо, умноженное на бесконечность, – это бесконечность, а что-либо, деленное на бесконечность, – ноль. В ECMAScript v1 и более поздних версиях вместо `Number.POSITIVE_INFINITY` можно также использовать предопределенную глобальную константу `Infinity`.

См. также

`Infinity`, `isFinite()`

Number.toExponential()

форматирует число в экспоненциальную форму представления

Синтаксис

число`.toExponential(`*разрядность*`)`

Аргументы

разрядность	Количество цифр после десятичной точки. Может быть значением от 0 до 20 включительно, конкретные реализации могут поддерживать больший диапазон значений. Если аргумент отсутствует, то цифр будет столько, сколько необходимо.

Возвращаемое значение

Строковое представление числа в экспоненциальной нотации с одной цифрой перед десятичной точкой и с количеством цифр, указанным в аргументе *разрядность*, после нее. Дробная часть, если это необходимо, округляется или дополняется нулями, чтобы она имела указанную длину.

Исключения

`RangeError`	Генерируется, если аргумент *разрядность* слишком велик или слишком мал. Значения между 0 и 20 включительно не приводят к ошибке `RangeError`. Реализациям также разрешено поддерживать большее или меньшее количество цифр.
`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Пример

```
var n = 12345.6789;
n.toExponential(1);  // Вернет 1.2e+4
n.toExponential(5);  // Вернет 1.23457e+4
n.toExponential(10); // Вернет 1.2345678900e+4
n.toExponential();   // Вернет 1.23456789e+4
```

См. также

`Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number.toString()`

Number.toFixed()

форматирует число в форму представления с фиксированной точкой

Синтаксис

число.`toFixed`(*разрядность*)

Аргументы

разрядность	Количество цифр после десятичной точки; оно может быть значением от 0 до 20 включительно; конкретные реализации могут поддерживать больший диапазон значений. Если этот аргумент отсутствует, он считается равным 0.

Возвращаемое значение

Строковое представление *числа*, которое не использует экспоненциальную нотацию и в котором количество цифр после десятичной точки равно аргументу *разрядность*. При необходимости число округляется, а дробная часть дополняется нулями до указанной длины. Если число больше, чем 1e+21, этот метод вызывает функцию `Number.toString()` и возвращает строку в экспоненциальной нотации.

Исключения

`RangeError`	Генерируется, если аргумент *разрядность* слишком велик или слишком мал. Значения от 0 до 20 включительно не приводят к исключению `RangeError`. Конкретные реализации могут поддерживать большие или меньшие значения.
`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Пример

```
var n = 12345.6789;
n.toFixed();            // Вернет 12346: обратите внимание на округление
                        // и отсутствие дробной части
n.toFixed(1);           // Вернет 12345.7: обратите внимание на округление
n.toFixed(6);           // Вернет 12345.678900: обратите внимание на добавление нулей
(1.23e+20).toFixed(2);  // Вернет 123000000000000000000.00
(1.23e-10).toFixed(2)   // Вернет 0.00
```

См. также

`Number.toExponential()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number. toString()`

Number.toLocaleString()

преобразует число в строку в соответствии с региональными настройками

Синтаксис

число.`toLocaleString()`

Возвращаемое значение

Зависящее от реализации строковое представление числа, отформатированное в соответствии с региональными настройками, на которое могут влиять, например, символы пунктуации, выступающие в качестве десятичной точки и разделителя тысяч.

Исключения

`TypeError` Генерируется, если метод вызван для объекта, не являющегося объектом `Number`.

См. также

`Number.toExponential()`, `Number.toFixed()`, `Number.toPrecision()`, `Number.toString()`

Number.toPrecision()

форматирует значащие цифры числа

Синтаксис

число.`toPrecision(`*точность*`)`

Аргументы

точность Количество значащих цифр в возвращаемой строке. Оно может быть значением от 1 до 21 включительно. Конкретные реализации могут поддерживать большие и меньшие значения *точности*. Если этот аргумент отсутствует, для преобразования в десятичное число используется метод `toString()`.

Возвращаемое значение

Строковое представление *числа*, содержащее количество значащих цифр, определяемое аргументом *точность*. Если *точность* имеет достаточно большое значение, чтобы включить все цифры целой части *числа*, возвращаемая строка записывается в нотации с фиксированной точкой. В противном случае запись осуществляется в экспоненциальной нотации с одной цифрой перед десятичной точкой и количеством цифр *точность*-1 после десятичной точки. Число при необходимости округляется или дополняется нулями.

Исключения

`RangeError`	Генерируется, если аргумент *точность* слишком мал или слишком велик. Значения от 1 до 21 включительно не приводят к исключению `RangeError`. Конкретные реализации могут поддерживать большие и меньшие значения.
`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Пример

```
var n = 12345.6789;
n.toPrecision(1);  // Вернет 1e+4
n.toPrecision(3);  // Вернет 1.23e+4
n.toPrecision(5);  // Вернет 12346: обратите внимание на округление
n.toPrecision(10); // Вернет 12345.67890: обратите внимание на добавление нуля
```

См. также

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toString()`

Number.toString()

преобразует число в строку **переопределяет Object.toString()**

Синтаксис

число.`toString`(*основание*)

Аргументы

основание	Необязательный аргумент, определяющий основание системы счисления (между 2 и 36), в которой должно быть представлено число. Если аргумент отсутствует, то основание равно 10. Следует заметить, что спецификация ECMAScript разрешает реализациям возвращать любое значение, если этот аргумент равен любому значению, отличному от 10.

Возвращаемое значение

Строковое представление числа.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

Описание

Метод `toString()` объекта `Number` преобразует число в строку. Если аргумент *основание* опущен или указано значение 10, число преобразуется в строку по основанию 10. Хотя спецификация ECMAScript не требует от реализаций корректно реагировать на любые другие значения аргумента *основание*, тем не менее все распространенные реализации принимают значения основания в диапазоне от 2 до 36.

См. также

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`

Number.valueOf()

преобразует число в строку **переопределяет Object.valueOf()**

Синтаксис

число.`valueOf()`

Возвращаемое значение

Элементарное числовое значение объекта `Number`. В явном вызове этого метода редко возникает необходимость.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `Number`.

См. также

`Object.valueOf()`

Object

надкласс, реализующий общие возможности всех JavaScript-объектов

Конструктор

`new Object()`

`new Object(`*значение*`)`

Аргументы

значение	Этот необязательный аргумент определяет элементарное значение – число, логическое значение или строку, которое должно быть преобразовано в объект `Number`, `Boolean` или `String`.

Возвращаемое значение

Если аргумент *значение* указан, конструктор возвращает вновь созданный экземпляр `Object`. Если указан аргумент *значение* элементарного типа, конструктор создаст объект-обертку `Number`, `Boolean` или `String` для указанного элементарного значения. Если конструктор `Object()` вызывается как функция (без оператора `new`), он действует точно так же, как при вызове с оператором `new`.

Свойства

`constructor`	Ссылка на функцию, которая была конструктором объекта.

Методы

`hasOwnProperty()`

Проверяет, имеет ли объект собственное (не унаследованное) свойство с указанным именем.

`isPrototypeOf()`

Проверяет, является ли данный объект прототипом для указанного объекта.

`propertyIsEnumerable()`

Проверяет, существует ли свойство с указанным именем и будет ли оно перечислено циклом `for/in`.

`toLocaleString()`

Возвращает локализованное строковое представление объекта. Реализация по умолчанию этого метода просто вызывает метод `toString()`, но подклассы могут переопределять его для выполнения локализации.

`toString()`

Возвращает строковое представление объекта. Реализация этого метода в классе `Object` является очень общей и возвращает немного полезной информации. Подклассы `Object` обычно переопределяют этот метод собственным методом `toString()`, возвращающим более полезный результат.

`valueOf()`

Возвращает элементарное значение объекта, если оно существует. Для объектов типа `Object` этот метод просто возвращает сам объект. Подклассы `Object`, такие как `Number` и `Boolean`, переопределяют этот метод, чтобы можно было получить элементарное значение, связанное с объектом.

Статические методы

В ECMAScript 5 конструктор `Object` служит пространством имен для следующих глобальных функций:

`Object.create()`

Создает новый объект с указанным прототипом и свойствами.

`Object.defineProperties()`

Создает или настраивает одно или более свойств в указанном объекте.

`Object.defineProperty()`

Создает или настраивает свойство в указанном объекте.

`Object.freeze()`

Делает указанный объект неизменяемым.

`Object.getOwnPropertyDescriptor()`

Возвращает атрибуты указанного свойства в указанном объекте.

`Object.getOwnPropertyNames()`

Возвращает массив имен всех неунаследованных свойств в указанном объекте, включая свойства, не перечисляемые циклом `for/in`.

`Object.getPrototypeOf()`

Возвращает прототип указанного объекта.

`Object.isExtensible()`

Определяет, могут ли добавляться новые свойства в указанный объект.

`Object.isFrozen()`

Определяет, является ли указанный объект фиксированным.

`Object.isSealed()`

Определяет, является ли указанный объект нерасширяемым, а его свойства недоступными для настройки.

`Object.keys()`

Возвращает массив имен неунаследованных перечислимых свойств в указанном объекте.

`Object.preventExtensions()`

Предотвращает возможность добавления новых свойств в указанный объект.

`Object.seal()`

Предотвращает возможность добавления новых и удаления существующих свойств в указанном объекте.

Описание

Класс `Object` – это встроенный тип данных языка JavaScript. Он играет роль надкласса для всех остальных JavaScript-объектов; следовательно, методы и поведение класса `Object` наследуются всеми остальными объектами. Об основных особенностях JavaScript-объектов рассказывается в главе 6.

В дополнение к показанному ранее конструктору `Object()` объекты могут создаваться и инициализироваться с помощью синтаксиса объектных литералов, описанного в разделе 6.1.

См. также

`Array`, `Boolean`, `Function`, `Function.prototype`, `Number`, `String`; глава 6

Object.constructor

функция-конструктор объекта

Синтаксис

объект`.constructor`

Описание

Свойство `constructor` любого объекта – это ссылка на функцию, являющуюся конструктором этого объекта. Например, если создать массив `a` с помощью конструктора `Array()`, то значением свойства `a.constructor` будет `Array`:

```
a = new Array(1,2,3);  // Создать объект
a.constructor == Array // Равно true
```

Одно из распространенных применений свойства `constructor` состоит в определении типа неизвестных объектов. Оператор `typeof` позволяет определить, является ли неизвестный объект элементарным значением или объектом. Если это объект, то посредством свойства `constructor` можно определить тип этого объекта. Например, следующая функция позволяет узнать, является ли данное значение массивом:

```
function isArray(x) {
    return ((typeof x == "object") && (x.constructor == Array));
}
```

Однако следует отметить, что, хотя этот прием эффективен для объектов, встроенных в базовый JavaScript, его работа с объектами среды выполнения клиентского JavaScript, такими как объект `Window`, не гарантируется. Реализация по умолчанию метода `Object.toString()` представляет другой способ определения типа неизвестного объекта.

См. также

`Object.toString()`

Object.create() ECMAScript 5

создает объект с указанным прототипом и свойствами

Синтаксис

`Object.create(`*прототип*`)`
`Object.create(`*прототип*`,` *дескрипторы*`)`

Аргументы

прототип	Прототип создаваемого объекта или `null`.
дескрипторы	Необязательный объект, отображающий имена свойств в их дескрипторы.

Возвращаемое значение

Вновь созданный объект, наследующий *прототип* и обладающий свойствами, описываемыми *дескрипторами*.

Исключения

`TypeError`	Генерируется, если *прототип* не является объектом или значением `null` или если указанные *дескрипторы* заставляют метод `Object.defineProperties()` сгенерировать исключение `TypeError`.

Описание

Функция `Object.create()` создает и возвращает новый объект с прототипом, определяемым аргументом *прототип*. Это означает, что новый объект наследует свойства от *прототипа*.

Если указан необязательный аргумент *дескрипторы*, функция `Object.create()` добавит в новый объект свойства, как если бы был вызван метод `Object.defineProperties()`. То есть вызов функции `Object.create(p,d)` с двумя аргументами эквивалентен вызовам:

```
Object.defineProperties(Object.create(p), d);
```

Дополнительную информацию об аргументе *дескрипторы* можно найти в справочной статье `Object.defineProperties()`, а описание дескрипторов свойств в справочной статье `Object.getOwnPropertyDescriptor()`.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
// Создать объект, который имеет собственные свойства x и y и наследует свойство z
var p = Object.create({z:0}, {
    x: { value: 1, writable: false, enumerable:true, configurable: true},
    y: { value: 2, writable: false, enumerable:true, configurable: true},
});
```

См. также

`Object.defineProperty()`, `Object.defineProperties()`, `Object.getOwnPropertyDescriptor()`, разделы 6.1, 6.7

Object.defineProperties() ECMAScript 5

создает или настраивает свойства объекта

Синтаксис

`Object.defineProperties(`*о*, *дескрипторы*`)`

Arguments

о	Объект, в котором будут создаваться или настраиваться свойства.
дескрипторы	Объект, отображающий имена свойств в их дескрипторы.

Возвращаемое значение

Объект *о*.

Исключения

`TypeError`	Генерируется, если аргумент *о* не является объектом или если какое-либо из указанных свойств не может быть создано или настроено. Эта функция не является атомарной: она может создать или настроить часть свойств и затем возбудить исключение, не создав или не настроив другие свойства. Перечень ошибок, которые могут вызвать исключение `TypeError`, приводится в разделе 6.7.

Описание

Функция `Object.defineProperties()` создает или настраивает свойства объекта *о*, указанные и описанные в аргументе *дескрипторы*. Имена свойств объекта *дескрипторы* являются именами свойств, которые будут созданы или настроены в объекте *о*, а значениями этих свойств являются объекты дескрипторов свойств, которые определяют атрибуты создаваемых или настраиваемых свойств.

Функция `Object.defineProperties()` действует подобно функции `Object.defineProperty()`; дополнительные подробности смотрите в описании этой функции. Дополнительные сведения об объектах дескрипторов свойств приводятся в справочной статье `Object.getOwnPropertyDescriptor()`.

Пример

```
// Добавить в новый объект свойства x и y, доступные только для чтения
var p = Object.defineProperties({}, {
    x: { value: 0, writable: false, enumerable:true, configurable: true},
    y: { value: 1, writable: false, enumerable:true, configurable: true},
});
```

См. также

`Object.create()`, `Object.defineProperty()`, `Object.getOwnPropertyDescriptor()`, раздел 6.7

Object.defineProperty() ECMAScript 5

создает или настраивает одно свойство в объекте

Синтаксис

`Object.defineProperty(`*о*, *имя*, *дескриптор*`)`

Аргументы

о	Объект, в котором будет создаваться или настраиваться свойство.

имя	Имя создаваемого или настраиваемого свойства.
дескриптор	Объект дескриптора свойства, описывающий новое свойство или изменения, которые должны быть выполнены в существующем свойстве.

Возвращаемое значение

Объект *o*.

Исключения

`TypeError`	Генерируется, если аргумент *o* не является объектом или если свойство не может быть создано (из-за того, что объект *o* является нерасширяемым) или настроено (например, из-за того, что уже существующее свойство является ненастраиваемым). Перечень ошибок, которые могут вызвать исключение `TypeError`, приводится в разделе 6.7.

Описание

Функция `Object.defineProperty()` создает или настраивает свойство с именем *имя* в объекте *o*, используя описание свойства в аргументе *дескриптор*. Дополнительные сведения об объектах дескрипторов свойств приводятся в справочной статье `Object.getOwnPropertyDescriptor()`.

Если объект *o* еще не имеет свойства с именем *имя*, эта функция просто создаст новое свойство с атрибутами и значением, указанными в *дескрипторе*. Если в *дескрипторе* не указаны какие-либо атрибуты, соответствующие им атрибуты получат значение `false` или `undefined`.

Если значение аргумента *имя* совпадает с именем существующего свойства объекта *o*, то функция `Object.defineProperty()` настроит это свойство, изменив его значение или атрибуты. В этом случае в *дескрипторе* достаточно указать только атрибуты, которые должны быть изменены: атрибуты, отсутствующие в *дескрипторе*, сохранят свои прежние значения.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
function constant(o, n, v) { // Определить константу o.n со значением v
    Object.defineProperty(o, n, { value: v, writable: false
                                  enumerable: true, configurable:false});
}
```

См. также

`Object.create()`, `Object.defineProperties()`, `Object.getOwnPropertyDescriptor()`, раздел 6.7

Object.freeze() ECMAScript 5

делает объект неизменяемым

Синтаксис

`Object.freeze(`*o*`)`

Аргументы

o	Объект, который должен быть зафиксирован.

Возвращаемое значение

Зафиксированный объект *o*.

Описание

Функция `Object.freeze()` делает объект *o* нерасширяемым (`Object.preventExtensions()`), а все его собственные свойства – ненастраиваемыми, подобно функции `Object.seal()`. Однако в дополнение к этому она делает все неунаследованные свойства доступными только для чтения. Это означает, что в объект *o* нельзя будет добавлять новые свойства, а существующие свойства-данные нельзя будет изменить или удалить. Действие функции `Object.freeze()` является необратимым, т.е. зафиксированный объект нельзя снова сделать доступным для изменения.

Имейте в виду, что функция `Object.freeze()` устанавливает атрибут `writable`, имеющийся только в свойствах-данных. Она не действует на свойства, имеющие методы записи. Отметьте также, что функция `Object.freeze()` не действует на унаследованные свойства.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

См. также

`Object.defineProperty()`, `Object.isFrozen()`, `Object.preventExtensions()`, `Object.seal()`, раздел 6.8.3

Object.getOwnPropertyDescriptor() ECMAScript 5

возвращает атрибуты свойства

Синтаксис

`Object.getOwnPropertyDescriptor(`*o*, *имя*`)`

Аргументы

o	Объект, которому принадлежит искомое свойство.
имя	Имя свойства (или индекс элемента массива), атрибуты которого требуется получить.

Возвращаемое значение

Объект дескриптора для указанного свойства заданного объекта или `undefined`, если такое свойство не существует.

Описание

Функция `Object.getOwnPropertyDescriptor()` возвращает дескриптор для указанного свойства заданного объекта. Дескриптор свойства – это объект, описывающий атрибуты и значение свойства. Более полная информация приводится в следующем подразделе. Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Дескрипторы свойств

Дескриптор свойства – это обычный JavaScript-объект, описывающий атрибуты (и иногда значение) свойства. В языке JavaScript существует два типа свойств. *Свойства-данные*, имеющие значение и три атрибута: `enumerable`, `writable` и `configurable`. *Свойства с методами доступа*, имеющие метод чтения и/или метод записи, а также атрибуты `enumerable` и `configurable`.

Дескриптор свойства с данными имеет следующий вид:

```
{
   value:        /* любое значение, допустимое в языке JavaScript */,
   writable:     /* true или false */,
   enumerable:   /* true или false */,
   configurable: /* true или false */
}
```

Дескриптор свойства с методами доступа имеет следующий вид:

```
{
   get:          /* функция или undefined: взамен свойства value */,
   set:          /* функция или undefined: взамен атрибута writable */,
   enumerable:   /* true или false */,
   configurable: /* true или false */
}
```

См. также

`Object.defineProperty()`, **раздел 6.7**

Object.getOwnPropertyNames() ECMAScript 5

возвращает имена неунаследованных свойств

Синтаксис

`Object.getOwnPropertyNames(o)`

Аргументы

o	**Объект.**

Возвращаемое значение

Массив, содержащий имена всех неунаследованных свойств объекта *o*, включая неперечислимые свойства.

Описание

Функция `Object.getOwnPropertyNames()` возвращает массив с именами всех неунаследованных объекта *o*, включая неперечислимые свойства. Для получения массива имен только перечислимых свойств можно использовать функцию `Object.keys()`.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
Object.getOwnPropertyNames([]) // => ["length"]: "length" - неперечислимое
```

См. также

`Object.keys()`, **раздел 6.5**

Object.getPrototypeOf() ECMAScript 5

возвращает прототип объекта

Синтаксис

`Object.getPrototypeOf(o)`

Аргументы

o Объект.

Возвращаемое значение

Прототип объекта *o*.

Описание

Функция `Object.getPrototypeOf()` возвращает прототип своего аргумента. Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
var p = {};               // Обычный объект
Object.getPrototypeOf(p) // => Object.prototype
var o = Object.create(p) // Объект, наследующий объект p
Object.getPrototypeOf(o) // => p
```

См. также

`Object.create()`; глава 6

Object.hasOwnProperty()

проверяет, является ли свойство унаследованным

Синтаксис

объект.`hasOwnProperty(`*имя_свойства*`)`

Аргументы

имя_свойства

Строка, содержащая имя свойства *объекта*.

Возвращаемое значение

Возвращает `true`, если *объект* имеет неунаследованное свойство с именем, заданным в *имени_свойства*. Возвращает `false`, если *объект* не имеет свойства с указанным именем или если он наследует это свойство от своего объекта-прототипа.

Описание

В главе 9 говорится, что JavaScript-объекты могут иметь собственные свойства, а также наследовать свойства от своих объектов-прототипов. Метод `hasOwnProperty()` предоставляет способ, позволяющий установить различия между унаследованными свойствами и неунаследованными локальными свойствами.

Пример

```
var o = new Object();            // Создать объект
o.x = 3.14;                      // Определить неунаследованное свойство
o.hasOwnProperty("x");           // Вернет true: x - это локальное свойство o
o.hasOwnProperty("y");           // Вернет false: o не имеет свойства y
o.hasOwnProperty("toString"); // Вернет false: свойство toString унаследовано
```

См. также

`Function.prototype`, `Object.propertyIsEnumerable()`; глава 9

Object.isExtensible() ECMAScript 5

возможно ли добавить в объект новое свойство?

Синтаксис

`Object.isExtensible(o)`

Аргументы

o Объект, проверяемый на возможность расширения

Возвращаемое значение

`true`, если в объект можно расширить новыми свойствами, и `false` – если нет.

Описание

Если в объект можно добавлять новые свойства, он является расширяемым. Все объекты сразу после создания являются расширяемыми и остаются таковыми, пока не будут переданы функции `Object.preventExtensions()`, `Object.seal()` или `Object.freeze()`.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
var o = {};                   // Создать новый объект
Object.isExtensible(o)        // => true: он является расширяемым
Object.preventExtensions(o);  // Сделать нерасширяемым
Object.isExtensible(o)        // => false: теперь он нерасширяемый
```

См. также

`Object.isFrozen()`, `Object.isSealed()`, `Object.preventExtensions()`, раздел 6.8.3

Object.isFrozen() ECMAScript 5

объект является неизменяемым?

Синтаксис

`Object.isFrozen(o)`

Аргументы

o Проверяемый объект.

Возвращаемое значение

`true`, если объект *o* является зафиксированным и неизменяемым, и `false` – если нет.

Описание

Объект считается зафиксированным, если все его неунаследованные свойства (кроме свойств с методами записи) доступны только для чтения и он является нерасширяемым. Объект считается нерасширяемым, если в него нельзя добавить новые (неунаследованные) свойства и из него нельзя удалить имеющиеся (неунаследованные) свойства. Функция `Object.isFrozen()` проверяет, является ли ее аргумент зафиксированным объектом или нет. Зафиксированный объект нельзя расфиксировать.

Обычно фиксация объектов выполняется с помощью функции `Object.freeze()`. Однако зафиксировать объект можно также с помощью функции `Object.preventExtensions()`

с последующим вызовом `Object.defineProperty()`, чтобы сделать все свойства объекта неудаляемыми и доступными только для чтения.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

См. также

`Object.defineProperty()`, `Object.freeze()`, `Object.isExtensible()`, `Object.isSealed()`, `Object.preventExtensions()`, `Object.seal()`, раздел 6.8.3

Object.isPrototypeOf()

проверяет, является ли один объект прототипом другого объекта

Синтаксис

объект.`isPrototypeOf(`*o*`)`

Аргументы

o Любой объект.

Возвращаемое значение

Возвращает `true`, если *объект* является прототипом объекта *o*. Возвращает `false`, если *o* не является объектом или если данный *объект* не является прототипом объекта *o*.

Описание

Как объяснялось в главе 9, объекты в языке JavaScript наследуют свойства от своих объектов-прототипов. К прототипу объекта можно обращаться с помощью свойства `prototype` функции-конструктора, которая использовалась для создания и инициализации объекта. Метод `isPrototypeOf()` позволяет определить, является ли один объект прототипом другого. Этот прием может применяться для определения класса объекта.

Пример

```
var o = new Object();                           // Создать объект
Object.prototype.isPrototypeOf(o)               // true: o - объект
Function.prototype.isPrototypeOf(o.toString); // true: toString - функция
Array.prototype.isPrototypeOf([1,2,3]);         // true: [1,2,3] - массив
// Ту же проверку можно выполнить другим способом
(o.constructor == Object); // true: o создан с помощью конструктора Object()
(o.toString.constructor == Function);           // true: o.toString - функция
// Объекты-прототипы сами имеют прототипы. Следующий вызов вернеттrue, показывая, что
// объекты-функции наследуют свойства от Function.prototype, а также от Object.prototype.
Object.prototype.isPrototypeOf(Function.prototype);
```

См. также

`Function.prototype`, `Object.constructor`; глава 9

Object.isSealed() ECMAScript 5

возможно ли добавлять в объект новые и удалять существующие свойства?

Синтаксис

`Object.isSealed(`*o*`)`

Аргументы

o Проверяемый объект.

Возвращаемое значение

`true`, если объект *o* является нерасширяемым, с недоступными для настройки свойствами, и `false` – если нет.

Описание

Объект считается нерасширяемым, с недоступными для настройки свойствами, если в него нельзя добавить новые (неунаследованные) свойства и нельзя удалить существующие (неунаследованные) свойства. Функция `Object.isSealed()` проверяет, является ли ее аргумент нерасширяемым объектом, с недоступными для настройки свойствами, или нет. Недоступные для настройки свойства нельзя вновь сделать настраиваемыми. Обычно такие объекты получают с помощью функции `Object.seal()` или `Object.freeze()`. Однако того же эффекта можно добиться с помощью функции `Object.preventExtensions()`, с последующим вызовом `Object.defineProperty()`, чтобы сделать все свойства объекта неудаляемыми.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

См. также

`Object.defineProperty()`, `Object.freeze()`, `Object.isExtensible()`, `Object.isFrozen()`, `Object.preventExtensions()`, `Object.seal()`, раздел 6.8.3

Object.keys() ECMAScript 5

возвращает имена собственных перечислимых свойств

Синтаксис

`Object.keys(o)`

Аргументы

o Объект.

Возвращаемое значение

Массив, содержащий имена всех перечислимых (неунаследованных) свойств объекта *o*.

Описание

Функция `Object.keys()` возвращает массив с именами свойств объекта *o*. Массив включает только имена свойств, которые являются перечислимыми и определены непосредственно в объекте *o*: унаследованные свойства не включаются. (Для получения имен неперечислимых свойств можно использовать функцию `Object.getOwnPropertyNames()`.) Свойства в массиве следуют в том же порядке, в каком они перечисляются циклом `for/in`.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

Пример

```
Object.keys({x:1, y:2}) // => ["x", "y"]
```

См. также

`Object.getOwnPropertyNames()`, разделы 5.5.4, 6.5

Object.preventExtensions() ECMAScript 5

предотвращает добавление в объект новых свойств

Синтаксис

`Object.preventExtensions(`*o*`)`

Аргументы

o Объект, который должен иметь расширяемый набор атрибутов.

Возвращаемое значение

Объект *o* с аргументами.

Описание

Функция `Object.preventExtensions()` присваивает значение `false` атрибуту `extensible` объекта *o*, вследствие чего в него нельзя будет добавлять новые свойства. Действие этой функции необратимо: нерасширяемый объект нельзя вновь сделать расширяемым.

Следует отметить, что `Object.preventExtensions()` не воздействует на цепочку прототипов, и нерасширяемый объект все еще можно расширить новыми наследуемыми свойствами.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

См. также

`Object.freeze()`, `Object.isExtensible()`, `Object.seal()`, раздел 6.8.3

Object.propertyIsEnumerable()

проверяет, будет ли свойство видимо для цикла for/in

Синтаксис

объект.`propertyIsEnumerable(`*имя_свойства*`)`

Аргументы

имя_свойства Строка, содержащая имя свойства объекта.

Возвращаемое значение

Возвращает `true`, если у *объекта* есть неунаследованное свойство с именем, указанным в аргументе *имя_свойства*, и если это свойство «перечислимое», т. е. оно может быть перечислено циклом `for/in` для *объекта*.

Описание

Инструкция `for/in` выполняет цикл по «перечислимым» свойствам объекта. Однако не все свойства объекта являются перечислимыми: свойства, добавленные в объект программным способом, перечислимы, а предопределенные свойства (например, методы) встроенных объектов обычно неперечислимы. Метод `propertyIsEnumerable()` позволяет установить различия между перечислимыми и неперечислимыми свойствами. Однако следует заметить: спецификация ECMAScript утверждает, что `propertyIs-`

`Enumerable()` не проверяет цепочку прототипов, т. е. этот метод годится только для локальных свойств объекта и не предоставляет способа для проверки перечисляемости унаследованных свойств.

Пример

```
var o = new Object();                // Создать объект
o.x = 3.14;                          // Определить свойство
o.propertyIsEnumerable("x");         // true: x - локальное и перечислимое
o.propertyIsEnumerable("y");         // false: o не имеет свойства y
o.propertyIsEnumerable("toString"); // false: toString унаследованное свойство
Object.prototype.propertyIsEnumerable("toString"); // false: неперечислимое
```

См. также

`Function.prototype`, `Object.hasOwnProperty()`; глава 6

Object.seal() ECMAScript 5

предотвращает добавление и удаление свойств

Синтаксис

`Object.seal(`*o*`)`

Аргументы

o Объект, который должен стать нерасширяемым, с недоступными для настройки свойствами.

Возвращаемое значение

Объект в аргументе *o*.

Описание

Функция `Object.seal()` делает объект *o* нерасширяемым (`Object.preventExtensions()`), а все его собственные свойства – ненастраиваемыми. Это предотвращает добавление новых свойств и удаление существующих. Действие этой функции необратимо: нерасширяемый объект с ненастраиваемыми свойствами нельзя вновь сделать расширяемым объектом.

Имейте в виду, что `Object.seal()` не делает свойства объекта доступными только для чтения; используйте для этого функцию `Object.freeze()`. Отметьте также, что `Object.seal()` не воздействует на унаследованные свойства. Если в цепочке прототипов объекта, обработанного функцией `Object.seal()`, имеется расширяемый и настраиваемый объект, тогда имеется возможность добавлять и удалять наследуемые им свойства.

Обратите внимание, что эта функция вызывается не как метод объекта: это глобальная функция, которая принимает объект в виде аргумента.

См. также

`Object.defineProperty()`, `Object.freeze()`, `Object.isSealed()`, `Object.preventExtensions()`, раздел 6.8.3

Object.toLocaleString()

возвращает локализованное строковое представление объекта

Синтаксис

объект`.toString()`

Возвращаемое значение

Строковое представление *объекта*.

Описание

Этот метод предназначен для получения строкового представления объекта, локализованного в соответствии с текущими региональными настройками. Метод `toLocaleString()`, предоставляемый по умолчанию классом `Object`, просто вызывает метод `toString()` и возвращает полученную от него нелокализованную строку. Однако обратите внимание, что другие классы, в том числе `Array`, `Date` и `Number`, определяют собственные версии этого метода для локализованного преобразования в строку. Вы также можете переопределить этот метод собственными классами.

См. также

`Array.toLocaleString()`, `Date.toLocaleString()`, `Number.toLocaleString()`, `Object.toString()`

Object.toString()

возвращает строковое представление объекта

Синтаксис

объект`.toString()`

Возвращаемое значение

Строка, представляющая *объект*.

Описание

Метод `toString()` относится к тем, которые обычно не вызываются явно в JavaScript-программах. Программист определяет этот метод в своих объектах, а система вызывает метод, когда требуется преобразовать объект в строку.

JavaScript вызывает метод `toString()` для преобразования объекта в строку всякий раз, когда объект используется в строковом контексте. Например, если объект преобразуется в строку при передаче в функцию, требующую строкового аргумента:

```
alert(my_object);
```

Подобным же образом объекты преобразуются в строки, когда они конкатенируются со строками с помощью оператора +:

```
var msg = 'Мой объект: ' + my_object;
```

Метод `toString()` вызывается без аргументов и должен возвращать строку. Чтобы от возвращаемой строки была какая-то польза, эта строка должна каким-либо образом базироваться на значении объекта, для которого был вызван метод.

Определяя в JavaScript специальный класс, целесообразно определить для него метод `toString()`. Если этого не сделать, объект наследует метод `toString()`, определенный по умолчанию в классе `Object`. Этот стандартный метод возвращает строку в формате:

```
[objectкласс]
```

где *класс* – это класс объекта: значение, такое как «Object», «String», «Number», «Function», «Window», «Document» и т. д. Такое поведение стандартного метода `toString()` иногда бывает полезно для определения типа или класса неизвестного объекта. Однако большинство объектов имеют собственную версию `toString()`, поэтому для произвольного объекта о необходимо явно вызывать метод `Object.toString()`, как показано ниже:

```
Object.prototype.toString.apply(o);
```

Обратите внимание, что этот способ идентификации неизвестных объектов годится только для встроенных объектов. Если вы определяете собственный класс объектов, то *класс* для него будет соответствовать значению «Object». В этом случае дополнительную информацию об объекте позволит получить свойство `Object.constructor`.

Метод `toString()` может быть очень полезен при отладке JavaScript-программ – он позволяет выводить объекты и видеть их значения. По одной только этой причине есть смысл определять метод `toString()` для каждого создаваемого вами класса.

Несмотря на то что метод `toString()` обычно вызывается системой автоматически, бывают случаи, когда его требуется вызвать явно. Например, чтобы выполнить явное преобразование объекта в строку, если JavaScript не делает это автоматически:

```
y = Math.sqrt(x);    // Вычислить число
ystr = y.toString(); // Преобразовать его в строку
```

Относительно этого примера следует помнить, что числа имеют встроенный метод `toString()`, обеспечивающий принудительное преобразование.

В других случаях вызов `toString()` может оказаться полезным – даже в таком контексте, когда JavaScript выполняет преобразование автоматически. Явное использование метода `toString()` может сделать программный код более понятным:

```
alert(my_obj.toString());
```

См. также

`Object.constructor()`, `Object.toLocaleString()`, `Object.valueOf()`

Object.valueOf()

элементарное значение указанного объекта

Синтаксис

объект.`valueOf()`

Возвращаемое значение

Элементарное значение, связанное с *объектом*, если оно есть. Если с *объектом* не связано значение, метод возвращает сам объект.

Описание

Метод `valueOf()` объекта возвращает элементарное значение, связанное с этим объектом, если оно есть. Для объектов типа `Object` элементарное значение отсутствует, и метод такого объекта возвращает сам объект.

Однако для объектов типа `Number` метод `valueOf()` возвращает элементарное числовое значение, представляемое объектом. Аналогично он возвращает элементарное логическое значение, связанное с объектом `Boolean`, или строку, связанную с объектом `String`.

Программисту редко приходится явно вызывать метод `valueOf()`. Интерпретатор JavaScript делает это автоматически всякий раз, когда встречает объект там, где ожидается элементарное значение. Из-за автоматического вызова метода `valueOf()` фактически трудно даже провести различие между элементарными значениями и соответствующими им объектами. Оператор `typeof`, например, показывает различие между строками и объектами `String`, но с практической точки зрения они работают в JavaScript-коде эквивалентным образом.

Метод `valueOf()` объектов `Number`, `Boolean` и `String` преобразует эти объекты-обертки в представляемые ими элементарные значения. Конструктор `Object()` выполняет противоположную операцию при вызове с числовым, логическим или строковым аргументом: он «заворачивает» элементарное значение в соответствующий объект-обертку. В большинстве случаев JavaScript берет это преобразование «элементарное значение – объект» на себя, поэтому необходимость в таком вызове конструктора `Object()` возникает редко.

Иногда программисту требуется определить специальный метод `valueOf()` для собственных объектов. Например, определить объектный JavaScript-тип для представления комплексных чисел (вещественное число плюс мнимое число). Как часть этого объектного типа, можно определить методы для выполнения комплексного сложения, умножения и т. д. Еще может потребоваться возможность рассматривать комплексные числа как обычные вещественные путем отбрасывания мнимой части. Для этого можно сделать примерно следующее:

```
Complex.prototype.valueOf = new Function("return this.real");
```

Определив метод `valueOf()` для собственного объектного типа `Complex`, можно, например, передавать объекты комплексных чисел в функцию `Math.sqrt()`, которая вычислит квадратный корень из вещественной части комплексного числа.

См. также

`Object.toString()`

parseFloat()

преобразует строку в число

Синтаксис

`parseFloat(`*`s`*`)`

Аргументы

s	Строка для синтаксического разбора, которая должна быть преобразована в число.

Возвращаемое значение

Возвращает выделенное из строки число или `NaN`, если *s* не начинается с допустимого числа. В JavaScript 1.0, если строка *s* не может быть преобразована в число, `parseFloat()` возвращает 0 вместо `NaN`.

Описание

Функция `parseFloat()` выполняет синтаксический разбор строки и возвращает первое число, найденное в *s*. Разбор прекращается и значение возвращается, когда `parseFloat()` встречает в *s* символ, который не является допустимой частью числа. Если *s* не начинается с числа, которое `parseFloat()` может разобрать, функция возвращает значение `NaN`. Проверка на это возвращаемое значение выполняется функцией `isNaN()`. Чтобы выделить только целую часть числа, используется функция `parseInt()`, а не `parseFloat()`.

См. также

`isNaN()`, `parseInt()`

parseInt()

преобразует строку в целое число

Синтаксис

`parseInt(s)`

`parseInt(s, основание)`

Аргументы

s	Строка для синтаксического разбора.
основание	Необязательный целочисленный аргумент, представляющий основание системы счисления анализируемого числа. Если этот аргумент отсутствует или равен 0, извлекается десятичное или шестнадцатеричное (если число начинается с префикса «0x» или «0X») число. Если этот аргумент меньше 2 или больше 36, `parseInt()` возвращает `NaN`.

Возвращаемое значение

Возвращается извлекаемое число (`NaN`, если *s* не начинается с корректного целого). В JavaScript 1.0, если невозможно выполнить синтаксический разбор строки *s*, `parseInt()` возвращает 0 вместо `NaN`.

Описание

Функция `parseInt()` выполняет синтаксический разбор строки *s* и возвращает первое число (с необязательным начальным знаком «минус»), найденное в *s*. Разбор останавливается и значение возвращается, когда `parseInt()` встречает в *s* символ, не являющийся допустимой цифрой для указанного *основания*. Если *s* не начинается с числа, которое может быть проанализировано функцией `parseInt()`, функция возвращает значение `NaN`. Проверка на это возвращаемое значение выполняется функцией `isNaN()`.

Аргумент *основание* задает основание извлекаемого числа. При основании, равном 10, `parseInt()` извлекает десятичное число. Если этот аргумент равен 8, то извлекается восьмеричное число (состоящее из цифр от 0 до 7), а если 16 – шестнадцатеричное (цифры от 0 до 9 и буквы от A до F). Аргумент *основание* может быть любым числом от 2 до 36.

Если *основание* равно 0 или не указано, `parseInt()` пытается определить систему счисления по строке *s*. Если *s* начинается (после необязательного знака «минус») с префикса «0x», `parseInt()` разбирает оставшуюся часть *s* как шестнадцатеричное число. Во всех остальных случаях `parseInt()` разбирает строку как десятичное число.

Пример

```
parseInt("19", 10); // Вернет 19 (10 + 9)
parseInt("11", 2);  // Вернет 3 (2 + 1)
parseInt("17", 8);  // Вернет 15 (8 + 7)
parseInt("1f", 16); // Вернет 31 (16 + 15)
parseInt("10");     // Вернет 10
parseInt("0x10");   // Вернет 16
```

См. также

`isNaN()`, `parseFloat()`

RangeError

генерируется, когда число выходит из допустимого диапазона **Object→Error→RangeError**

Конструктор

`new RangeError()`

`new RangeError(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он используется в качестве значения свойства `message` объекта `RangeError`.

Возвращаемое значение

Вновь созданный объект `RangeError`. Если указан аргумент *сообщение*, то для объекта `RangeError` он будет выступать в качестве значения свойства `message`; в противном случае `RangeError` возьмет в качестве значения этого свойства строку по умолчанию, определенную в реализации. Конструктор `RangeError()`, вызываемый как функция (без оператора `new`), ведет себя так же, как и при вызове с оператором `new`.

Свойства

`message`	Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или предлагаемую по умолчанию строку, определенную в реализации. Дополнительные сведения см. в справочной статье `Error.message`.
`name`	Строка, определяющая тип исключения. Все объекты `RangeError` наследуют для этого свойства строку «RangeError».

Описание

Экземпляр класса `RangeError` создается, когда числовое значение оказывается вне допустимого диапазона. Например, установка длины массива равной отрицательному числу приводит к генерации исключения `RangeError`. Дополнительные сведения о генерации и перехвате исключений см. в справочной статье `Error`.

См. также

`Error`, `Error.message`, `Error.name`

ReferenceError

генерируется при попытке чтения несуществующей переменной **Object→Error→ReferenceError**

Конструктор

`new ReferenceError()`

`new ReferenceError(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта `ReferenceError`.

Возвращаемое значение

Вновь созданный объект `ReferenceError`. Если указан аргумент *сообщение*, объект `ReferenceError` берет его в качестве значения своего свойства `message`; в противном случае он берет строку по умолчанию, определенную в реализации. Конструктор `ReferenceError()`, вызываемый как функция (без оператора `new`), ведет себя так же, как при вызове с оператором `new`.

Свойства

`message`	Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Дополнительные сведения см. в справочной статье `Error.message`.
`name`	Строка, определяющая тип исключения. Все объекты `ReferenceError` наследуют для этого свойства строку «ReferenceError».

Описание

Экземпляр класса `ReferenceError` создается при попытке прочитать значение несуществующей переменной. Дополнительные сведения о генерации и перехвате исключений см. в справочной статье `Error`.

См. также

`Error`, `Error.message`, `Error.name`

RegExp

регулярные выражения для поиска по шаблону **Object→RegExp**

Синтаксис литерала

/*маска*/*атрибуты*

Конструктор

`new RegExp(`*шаблон*, *атрибуты*`)`

Аргументы

шаблон	Строка, задающая шаблон регулярного выражения или другое регулярное выражение.
атрибуты	Необязательная строка, содержащая любые из атрибутов «g», «i» и «m», задающих глобальный, нечувствительный к регистру и многострочный поиск, соответственно. До выхода стандарта ECMAScript атрибут «m» не был доступен. Если аргумент *шаблон* – это регулярное выражение, а не строка, аргумент *атрибуты* может отсутствовать.

Возвращаемое значение

Возвращается новый объект `RegExp` с указанными шаблоном и атрибутами. Если аргумент *шаблон* представляет собой регулярное выражение, а не строку, конструктор `RegExp()` создаст новый объект `RegExp`, используя тот же шаблон и атрибуты, что и в указанном объекте `RegExp`. Если `RegExp()` вызывается как функция (без оператора `new`), то ведет себя так же, как при вызове с оператором `new`, кроме случая, когда *шаблон* уже является объектом `RegExp`; тогда функция возвращает аргумент *шаблон*, а не создает новый объект `RegExp`.

Исключения

`SyntaxError`	Генерируется, если *шаблон* не является допустимым регулярным выражением или если аргумент *атрибуты* содержит символы, отличные от «g», «i» и «m».
`TypeError`	Генерируется, если *шаблон* – это объект `RegExp` и аргумент *атрибуты* не опущен.

Свойства экземпляра

`global`	Признак присутствия в `RegExp` атрибута «g».
`ignoreCase`	Признак присутствия в `RegExp` атрибута «i».
`lastIndex`	Позиция символа при последнем обнаружении соответствия; используется для поиска в строке нескольких соответствий.
`multiline`	Признак присутствия в `RegExp` атрибута «m».
`source`	Исходный текст регулярного выражения.

Методы

`exec()`	Выполняет мощный универсальный поиск по шаблону.
`test()`	Проверяет, содержит ли строка данный шаблон.

Описание

Объект `RegExp` представляет регулярное выражение – мощное средство для поиска в строках по шаблону. Синтаксис и применение регулярных выражений полностью описаны в главе 10.

См. также

Глава 10

RegExp.exec()

универсальный поиск по шаблону

Синтаксис

regexp.`exec`(*строка*)

Аргументы

строка	Строка, в которой выполняется поиск.

Возвращаемое значение

Массив, содержащий результаты поиска или значение `null`, если соответствия не найдено. Формат возвращаемого массива описан далее.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом RegExp.

Описание

Метод `exec()` – наиболее мощный из всех методов объектов `RegExp` и `String` для поиска по шаблону. Это универсальный метод, использовать который несколько сложнее, чем методы `RegExp.test()`, `String.search()`, `String.replace()` и `String.match()`.

Метод `exec()` ищет в *строке* текст, соответствующий выражению *regexp*. И если находит, то возвращает массив результатов; в противном случае возвращается значение `null`. Элемент 0 полученного массива представляет собою искомый текст. Элемент 1 – это текст, соответствующий первому подвыражению в скобках внутри *regexp*, если оно есть. Элемент 2 соответствует второму подвыражению и т. д. Свойство `length` массива, как обычно, определяет количество элементов в массиве. В дополнение к элементам массива и свойству `length` значение, возвращаемое `exec()`, имеет еще два свойства. Свойство `index` указывает позицию первого символа искомого текста. Свойство `input` ссылается на *строку*. Этот возвращаемый массив совпадает с массивом, возвращаемым методом `String.match()`, когда он вызывается для неглобального объекта `RegExp`.

Когда метод `exec()` вызывается для неглобального шаблона, он выполняет поиск и возвращает описанный выше результат. Однако если *regexp* – глобальное регулярное выражение, `exec()` ведет себя несколько сложнее. Он начинает поиск в строке с символьной позиции, заданной свойством *regexp*.`lastIndex`. Найдя соответствие, метод устанавливает свойство `lastIndex` равным позиции первого символа после найденного соответствия. Это значит, что `exec()` можно вызвать несколько раз, чтобы выполнить цикл по всем соответствиям в строке. Если метод `exec()` больше не находит соответствий, он возвращает значение `null` и сбрасывает свойство `lastIndex` в ноль. Начиная поиск непосредственно после успешного нахождения соответствия в другой строке, необходимо соблюдать внимательность и вручную установить свойство `lastIndex` равным нулю.

Обратите внимание: `exec()` всегда включает полную информацию для найденного соответствия в возвращаемый им массив независимо от того, является *regexp* глобальным шаблоном или нет. Этим `exec()` отличается от метода `String.match()`, который возвращает намного меньше информации при работе с глобальными шаблонами. Вызов `exec()` в цикле – единственный способ получить полную информацию о результатах поиска для глобального шаблона.

Пример

Для нахождения всех соответствий в строке метод `exec()` можно вызывать в цикле:

```
var pattern = /\bJava\w*\b/g;
var text = "JavaScript is more fun than Java or JavaBeans!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched '" + result[0] +
          "' at position " + result.index +
          " next search begins at position " + pattern.lastIndex);
}
```

См. также

`RegExp.lastIndex`, `RegExp.test()`, `String.match()`, `String.replace()`, `String.search()`; глава 10

RegExp.global

выполняется ли глобальный поиск по данному регулярному выражению

Синтаксис

regexp.`global`

Описание

`global` – это логическое свойство объектов `RegExp`, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение глобальный поиск, т. е. было ли оно создано с атрибутом «g».

RegExp.ignoreCase

чувствительно ли регулярное выражение к регистру

Синтаксис

regexp.`ignoreCase`

Описание

`ignoreCase` – это логическое свойство объектов `RegExp`, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение поиск без учета регистра, т. е. было ли оно создано с атрибутом «i».

RegExp.lastIndex

начальная позиция следующего поиска

Синтаксис

regexp.`lastIndex`

Описание

`lastIndex` – это доступное для чтения и записи свойство объектов `RegExp`. Для регулярных выражений с установленным атрибутом «g» оно содержит целое, указывающее позицию в строке символа, который следует непосредственно за последним соответствием, найденным с помощью методов `RegExp.exec()` и `RegExp.test()`. Эти методы используют данное свойство в качестве начальной точки при следующем поиске. Благодаря этому данные методы можно вызывать повторно для выполнения цикла по всем соответствиям в строке. Обратите внимание: `lastIndex` не используется объектами `RegExp`, не имеющими атрибута «g» и не представляющими собой глобальные шаблоны.

Это свойство доступно для чтения и для записи, поэтому можно установить его в любой момент, чтобы указать, где в целевой строке должен быть начат следующий поиск. Методы `exec()` и `test()` автоматически сбрасывают свойство `lastIndex` в 0, когда не могут найти какого-либо (или следующего) соответствия. Начиная поиск в новой строке после успешного поиска в предыдущей, необходимо явно установить это свойство равным 0.

См. также

`RegExp.exec()`, `RegExp.test()`

RegExp.source

текст регулярного выражения

Синтаксис

regexp.`source`

Описание

`source` – доступное только для чтения строковое свойство объектов `RegExp`, содержащее текст шаблона `RegExp`. Текст не включает ограничивающие символы слэша, используемые в литералах регулярных выражений, а также не включает атрибуты «g», «i» и «m».

RegExp.test()

проверяет, соответствует ли строка шаблону

Синтаксис

regexp.`test`(*строка*)

Аргументы

строка	Проверяемая строка.

Возвращаемое значение

Возвращает `true`, если *строка* содержит текст, соответствующий *regexp*, и `false` – в противном случае.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `RegExp`.

Описание

Метод `test()` проверяет *строку*, чтобы увидеть, содержит ли она текст, который соответствует *regexp*. Если да, он возвращает `true`, в противном случае – `false`. Вызов метода `test()` для регулярного выражения *r* и передача ему строки *s* эквивалентны следующему выражению:

```
(r.exec(s) != null)
```

Пример

```
var pattern = /java/i;
pattern.test("JavaScript"); // Вернет true
pattern.test("ECMAScript"); // Вернет false
```

См. также

`RegExp.exec()`, `RegExp.lastIndex`, `String.match()`, `String.replace()`, `String.substring()`; глава 10

RegExp.toString()

преобразует регулярное выражение в строку **переопределяет Object.toString()**

Синтаксис

regexp.`toString()`

Возвращаемое значение

Строковое представление *regexp*.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, который не является объектом `RegExp`.

Описание

Метод `RegExp.toString()` возвращает строковое представление регулярного выражения в форме литерала регулярного выражения.

Обратите внимание: от реализаций не требуется обязательного добавления управляющих последовательностей, гарантирующих, что возвращаемая строка будет корректным литералом регулярных выражений. Рассмотрим регулярное выражение, созданное с помощью конструкции new `RegExp("/", "g")`. Реализация `RegExp.toString()` может вернуть для регулярного выражения `///g` либо добавить управляющую последовательность и вернуть `/\//g`.

String

поддержка строк **Object→String**

Конструктор

```
new String(s) // Функция-конструктор
String(s)     // Функция преобразования
```

Аргументы

s	Значение, подлежащее сохранению в объекте `String` или преобразованию в элементарное строковое значение.

Возвращаемое значение

Когда функция `String()` вызывается в качестве конструктора (с оператором `new`), она возвращает объект `String`, содержащий строку *s* или строковое представление *s*. Конструктор `String()`, вызванный без оператора `new`, преобразует *s* в элементарное строковое значение и возвращает преобразованное значение.

Свойства

`length`	Количество символов в строке.

Методы

`charAt()`	Извлекает из строки символ, находящийся в указанной позиции.
`charCodeAt()`	Возвращает код символа, находящегося в указанной позиции.
`concat()`	Выполняет конкатенацию одного или нескольких значений со строкой.
`indexOf()`	Выполняет поиск символа или подстроки в строке.
`lastIndexOf()`	Выполняет поиск символа или подстроки в строке с конца.
`localeCompare()`	Сравнивает строки с учетом порядка следования символов национальных алфавитов.
`match()`	Выполняет поиск по шаблону с помощью регулярного выражения.
`replace()`	Выполняет операцию поиска и замены с помощью регулярного выражения.
`search()`	Ищет в строке подстроку, соответствующую регулярному выражению.
`slice()`	Возвращает фрагмент строки или подстроку в строке.
`split()`	Разбивает строку на массив строк по указанной строке-разделителю или регулярному выражению.
`substr()`	Извлекает подстроку из строки. Аналог метода `substring()`.
`substring()`	Извлекает подстроку из строки.

`toLowerCase()`	Возвращает копию строки, в которой все символы переведены в нижний регистр.
`toString()`	Возвращает элементарное строковое значение.
`toUpperCase()`	Возвращает копию строки, в которой все символы переведены в верхний регистр.
`trim()`	Возвращает копию строки, из которой удалены все начальные и конечные пробельные символы.
`valueOf()`	Возвращает элементарное строковое значение.

Статические методы

`String.fromCharCode()`

Создает новую строку, помещая в нее принятые в качестве аргументов коды символов.

HTML-методы

С первых дней создания JavaScript в классе `String` определено несколько методов, которые возвращают строку, измененную путем добавления к ней HTML-тегов. Эти методы никогда не были стандартизованы в ECMAScript, но они позволяют динамически генерировать разметку HTML и в клиентских, и в серверных сценариях на языке JavaScript. Если вы готовы к использованию нестандартных методов, можете следующим образом создать разметку HTML для гиперссылки, выделенной полужирным шрифтом красного цвета:

```
var s = "щелкни здесь!";
var html = s.bold().link("JavaScript:alert('hello')").fontcolor("red");
```

Поскольку эти методы не стандартизованы, для них отсутствуют отдельные справочные статьи:

`anchor(`*имя*`)`	Возвращает копию строки в окружении тега `<a name=>`.
`big()`	Возвращает копию строки в окружении тега `<big>`.
`blink()`	Возвращает копию строки в окружении тега `<blink>`.
`bold()`	Возвращает копию строки в окружении тега `<b>`.
`fixed()`	Возвращает копию строки в окружении тега `<tt>`.
`fontcolor(`*цвет*`)`	Возвращает копию строки в окружении тега `<font color=>`.
`fontsize(`*размер*`)`	Возвращает копию строки в окружении тега `<font size=>`.
`italics()`	Возвращает копию строки в окружении тега `<i>`.
`link(`*url*`)`	Возвращает копию строки в окружении тега `<a href=>`.
`small()`	Возвращает копию строки в окружении тега `<small>`.
`strike()`	Возвращает копию строки в окружении тега `<strike>`.
`sub()`	Возвращает копию строки в окружении тега `<sub>`.
`sup()`	Возвращает копию строки в окружении тега `<sup>`.

Описание

Строки – это элементарный тип данных в JavaScript. Класс `String` предоставляет методы для работы с элементарными строковыми значениями. Свойство `length` объекта `String` указывает количество символов в строке. Класс `String` определяет немало методов для работы со строками. Например, имеются методы для извлечения символа или подстроки из строки или для поиска символа или подстроки. Обратите внимание:

строки JavaScript *не изменяются* – ни один из методов, определенных в классе `String`, не позволяет изменять содержимое строки. Зато методы, подобные `String.toUpperCase()`, возвращают абсолютно новую строку, не изменяя исходную.

В ECMAScript 5 и во многих реализациях JavaScript, вышедших до ES5, строки ведут себя как массивы символов, доступные только для чтения. Например, чтобы извлечь третий символ из строки `s`, можно написать `s[2]` вместо `s.charAt(2)`. Кроме того, инструкция `for/in`, примененная к строке, позволяет перечислить индексы массива для каждого символа в строке.

См. также

Глава 3

String.charAt()

возвращает n-й символ строки

Синтаксис

строка.`charAt(`*n*`)`

Аргументы

n Индекс символа, который должен быть извлечен из строки.

Возвращаемое значение

n-й символ строки.

Описание

Метод `String.charAt()` возвращает *n*-й символ строки. Номер первого символа в строке равен нулю. Если *n* не находится между 0 и *строка*.`length-1`, этот метод возвращает пустую строку. Обратите внимание: в JavaScript нет символьного типа данных, отличного от строкового, поэтому извлеченный символ представляет собой строку длиной 1.

См. также

`String.charCodeAt()`, `String.indexOf()`, `String.lastIndexOf()`

String.charCodeAt()

возвращает код n-го символа строки

Синтаксис

строка.`charCodeAt(`*n*`)`

Аргументы

n Индекс символа, код которого должен быть получен.

Возвращаемое значение

Код Юникода *n*-го символа в *строке* – 16-разрядное целое между 0 и 65 535.

Описание

Метод `charCodeAt()` аналогичен методу `charAt()`, за исключением того, что возвращает код символа, находящегося в определенной позиции, а не подстроку, содержащую сам символ. Если значение *n* отрицательно либо меньше или равно длине строки, `charCodeAt()` возвращает `NaN`.

Создание строки по коду Юникода символа описано в справочной статье `String.fromCharCode()`.

См. также

`String.charAt()`, `String.fromCharCode()`

String.concat()

объединяет строки

Синтаксис

строка.`concat`(*значение*, ...)

Аргументы

значение,	Одно или более значений, объединяемых со строкой.

Возвращаемое значение

Новая строка, полученная при объединении всех аргументов со строкой.

Описание

`concat()` преобразует все свои аргументы в строки (если это нужно) и добавляет их по порядку в конец *строки*. Возвращает полученную объединенную строку. Обратите внимание: сама *строка* при этом не изменяется.

Метод `String.concat()` представляет собой аналог метода `Array.concat()`. Следует отметить, что конкатенацию строк часто проще выполнить с помощью оператора +.

См. также

`Array.concat()`

String.fromCharCode()

создает строку из кодов символов

Синтаксис

`String.fromCharCode`(*c1*, *c2*, ...)

Аргументы

c1, *c2*, ...	Ноль или более целых значений, определяющих коды Юникода для символов создаваемой строки.

Возвращаемое значение

Новая строка, содержащая символы с указанными кодами.

Описание

Этот статический метод обеспечивает создание строки из отдельных числовых кодов Юникода ее символов, заданных в качестве аргументов. Следует заметить, что статический метод `fromCharCode()` является свойством конструктора `String()` и фактически не является строковым методом или методом объектов `String`.

Парным для описываемого метода является метод экземпляра `String.charCodeAt()`, который служит для получения кодов отдельных символов строки.

Пример

```
// Создать строку "hello"
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

См. также

`String.charCodeAt()`

String.indexOf()

поиск подстроки в строке

Синтаксис

строка.`indexOf(`*подстрока*`)`

строка.`indexOf(`*подстрока*, *начало*`)`

Аргументы

подстрока	Подстрока, которая должна быть найдена в *строке*.
начало	Необязательный целый аргумент, задающий позицию в строке, с которой следует начать поиск. Допустимые значения от 0 (позиция первого символа в строке) до *строка*.`length-1` (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с первого символа строки.

Возвращаемое значение

Позиция первого вхождения *подстроки* в *строку*, начиная с позиции *начало*, если подстрока найдена, или -1, если не найдена.

Описание

`String.indexOf()` выполняет поиск в *строке* от начала к концу, чтобы увидеть, содержит ли она искомую *подстроку*. Поиск начинается с позиции *начало* в строке или с начала строки, если аргумент *начало* не указан. Если *подстрока* найдена, `String.indexOf()` возвращает позицию первого символа первого вхождения *подстроки* в *строку*. Позиции символов в *строке* нумеруются с нуля.

Если *подстрока* в *строке* не найдена, `String.indexOf()` возвращает -1.

См. также

`String.charAt()`, `String.lastIndexOf()`, `String.substring()`

String.lastIndexOf()

поиск подстроки в строке, начиная с конца

Синтаксис

строка.`lastIndexOf(`*подстрока*`)`

строка.`lastIndexOf(`*подстрока*, *начало*`)`

Аргументы

подстрока	Подстрока, которая должна быть найдена в *строке*.

начало Необязательный целый аргумент, задающий позицию в строке, с которой следует начать поиск. Допустимые значения: от 0 (позиция первого символа в строке) до *строка*.`length-1` (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с последнего символа строки.

Возвращаемое значение

Позиция последнего вхождения подстроки в строку, начиная с позиции *начало*, если *подстрока* найдена, или -1, если такое вхождение не найдено.

Описание

`String.lastIndexOf()` просматривает строку от конца к началу, чтобы увидеть, содержит ли она *подстроку*. Поиск выполняется с позиции *начало* внутри *строки* или с конца *строки*, если аргумент *начало* не указан. Если *подстрока* найдена, `String.lastIndexOf()` возвращает позицию первого символа этого вхождения. Метод выполняет поиск от конца к началу, поэтому первое найденное вхождение является последним в строке, расположенным до позиции *начало*.

Если *подстрока* не найдена, `String.lastIndexOf()` возвращает -1.

Обратите внимание: хотя метод `String.lastIndexOf()` ищет строку от конца к началу, он все равно нумерует позиции символов в строке с начала. Первый символ строки занимает позицию с номером 0, а последний – *строка*.`length-1`.

См. также

`String.charAt()`, `String.indexOf()`, `String.substring()`

String.length

длина строки

Синтаксис

строка.`length`

Описание

Свойство `String.length` – это доступное только для чтения целое, указывающее количество символов в *строке*. Для любой строки `s` индекс последнего символа равен `s.length-1`. Свойство `length` строки не перечисляется циклом `for/in` и не может быть удалено с помощью оператора `delete`.

String.localeCompare()

сравнивает строки с учетом порядка следования символов национальных алфавитов

Синтаксис

строка.`localeCompare(`*целевая_строка*`)`

Аргументы

целевая_строка

Строка, сравниваемая со *строкой* с учетом порядка следования символов национальных алфавитов.

Возвращаемое значение

Число, обозначающее результат сравнения. Если *строка* «меньше» *целевой_строки*, `localeCompare()` возвращает отрицательное число. Если *строка* «больше» *целевой_строки*, метод возвращает положительное число. Если строки идентичны или неразличимы в соответствии с региональными соглашениями о сортировке, метод возвращает 0.

Описание

Когда к строкам применяются операторы < и >, сравнение выполняется только по кодам Юникода этих символов; порядок сортировки, принятый в текущем регионе, не учитывается. Сортировка, выполняемая подобным образом, не всегда оказывается верной. Возьмем, например, испанский язык, в котором буквы «ch» традиционно сортируются как одна буква, расположенная между буквами «c» и «d».

Метод `localeCompare()` служит для сравнения строк с учетом порядка сортировки, по умолчанию определяемого региональными настройками. Стандарт ECMAScript не определяет, как должно выполняться сравнение с учетом региона; в нем просто указано, что эта функция руководствуется порядком сортировки, определенным операционной системой.

Пример

Отсортировать массив строк в порядке, учитывающем региональные параметры, можно следующим образом:

```
var strings; // Сортируемый массив строк; инициализируется в другом месте
strings.sort(function(a,b) { return a.localeCompare(b) });
```

String.match()

находит одно или более соответствий регулярному выражению

Синтаксис

строка.`match`(*regexp*)

Аргументы

regexp	Объект `RegExp`, задающий шаблон для поиска. Если этот аргумент не является объектом `RegExp`, он сначала преобразуется с помощью конструктора `RegExp()`.

Возвращаемое значение

Массив, содержащий результаты поиска. Содержимое массива зависит от того, установлен ли в *regexp* глобальный атрибут «g». Далее это возвращаемое значение описано подробно.

Описание

Метод `match()` ищет в *строке* соответствия шаблону, определяемому выражением *regexp*. Поведение этого метода существенно зависит от того, установлен атрибут «g» в *regexp* или нет (регулярные выражения подробно рассмотрены в главе 10).

Если в *regexp* нет атрибута «g», `match()` ищет одно соответствие в *строке*. Если соответствие не найдено, `match()` возвращает `null`. В противном случае метод возвращает массив, содержащий информацию о найденном соответствии. Элемент массива с индексом 0 содержит найденный текст. Оставшиеся элементы содержат текст, соответствующий всем подвыражениям внутри регулярного выражения. В дополнение к этим обычным элементам массива возвращаемый массив имеет еще два объектных свойства. Свойство `index` массива указывает позицию начала найденного текста внутри *строки*. Кроме того, свойство `input` возвращаемого массива содержит ссылку на саму *строку*.

Если в *regexp* установлен флаг «g», `match()` выполняет глобальный поиск, находя в *строке* все соответствующие подстроки. Метод возвращает `null`, если соответствие не найдено, или массив, если найдено одно и более соответствий. Однако содержимое полученного массива при глобальном поиске существенно отличается. В этом случае элементы массива содержат все подстроки, найденные в *строке*, а сам массив не имеет свойств `index` и `input`. Обратите внимание: для глобального поиска `match()` не предоставляет информации о подвыражениях в скобках и не указывает, в каком месте *строки* было найдено каждое из соответствий. Эту информацию для глобального поиска можно получить методом `RegExp.exec()`.

Пример

Следующий фрагмент реализует глобальный поиск и находит все числа в строке:

```
"1 plus 2 equals 3".match(/\d+/g) // Вернет ["1", "2", "3"]
```

Следующий фрагмент реализует неглобальный поиск и использует более сложное регулярное выражение с несколькими подвыражениями. Он находит URL-адрес, а подвыражения регулярного выражения соответствуют протоколу, хосту и пути в URL:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];  // Содержит "http://www.isp.com/~david"
    var protocol = result[1]; // Содержит "http"
    var host = result[2];     // Содержит "www.isp.com"
    var path = result[3];     // Содержит "~david"
}
```

См. также

`RegExp`, `RegExp.exec()`, `RegExp.test()`, `String.replace()`, `String.search()`; глава 10

String.replace()

заменяет подстроку (подстроки), соответствующую регулярному выражению

Синтаксис

строка.`replace`(*regexp*, *замена*)

Аргументы

regexp	Объект `RegExp`, определяющий шаблон заменяемой подстроки. Если этот аргумент представляет собой строку, то она выступает в качестве текстового шаблона для поиска и не преобразуется в объект `RegExp`.
замена	Строка, определяющая текст замены, или функция, вызываемая для генерации текста замены. Подробности см. в подразделе «Описание».

Возвращаемое значение

Новая строка, в которой первое или все соответствия регулярному выражению *regexp* заменены строкой *замена*.

Описание

Метод `replace()` выполняет операцию поиска и замены для строки. Он ищет в *строке* одну или несколько подстрок, соответствующих регулярному выражению *regexp*, и заменяет их значением аргумента *замена*. Если в *regexp* указан глобальный атрибут «g», `replace()` заменяет все найденные подстроки. В противном случае метод заменяет только первую найденную подстроку.

Параметр *замена* может быть либо строкой, либо функцией. Если это строка, то каждое найденное соответствие заменяется указанной строкой. Имейте в виду, что символ $ имеет особый смысл в строке *замена*. Как показано в следующей таблице, он показывает, что для замены используется строка, производная от найденного соответствия.

Символы	Замена
$1, $2, ..., $99	Текст, соответствующий подвыражению с номером от 1 до 99 внутри регулярного выражения *regexp*
$&	Подстрока, соответствующая *regexp*
$`	Текст слева от найденной подстроки
$'	Текст справа от найденной подстроки
$$	Символ доллара

В стандарте ECMAScript v3 определено, что аргумент *замена* метода `replace()` может быть функцией, а не строкой. В этом случае функция будет вызываться для каждого найденного соответствия, а возвращаемая ею строка будет использоваться в качестве текста для замены. Первый аргумент, передаваемый функции, представляет собой строку, соответствующую шаблону. Следующие за ним аргументы – это строки, соответствующие любым подвыражениям внутри шаблона. Таких аргументов может быть ноль или более. Следующий аргумент – это целое, указывающее позицию внутри строки, в которой было найдено соответствие, а последний аргумент функции *замена* – это сама *строка*.

Пример

Обеспечение правильного регистра букв в слове «JavaScript»:

```
text.replace(/JavaScript/i, "JavaScript");
```

Преобразование имени из формата «Doe, John» в формат «John Doe»:

```
name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");
```

Замена всех двойных кавычек двумя одинарными закрывающими и двумя одинарными открывающими кавычками:

```
text.replace(/"([^"]*)"/g, "``$1''");
```

Перевод первых букв всех слов в строке в верхний регистр:

```
text.replace(/\b\w+\b/g, function(word) {
                           return word.substring(0,1).toUpperCase() +
                                  word.substring(1);
                         });
```

См. также

`RegExp`, `RegExp.exec()`, `RegExp.test()`, `String.match()`, `String.search()`; глава 10

String.search()

поиск соответствия регулярному выражению

Синтаксис

строка.`search(`*regexp*`)`

Аргументы

regexp	Объект `RegExp`, определяющий шаблон, который будет использоваться для поиска в *строке*. Если этот аргумент не является объектом `RegExp`, он сначала преобразуется путем передачи его конструктору `RegExp()`.

Возвращаемое значение

Позиция начала первой подстроки в *строке*, соответствующей выражению *regexp*, или -1, если соответствие не найдено.

Описание

Метод `search()` ищет подстроку в *строке*, соответствующую регулярному выражению *regexp*, и возвращает позицию первого символа найденной подстроки или -1, если соответствие не найдено.

Метод не выполняет глобального поиска, игнорируя флаг «g». Он также игнорирует свойство *regexp*.`lastIndex` и всегда выполняет поиск с начала строки, следовательно, всегда возвращает позицию первого соответствия, найденного в строке.

Пример

```
var s = "JavaScript is fun";
s.search(/script/i) // Вернет 4
s.search(/a(.)a/)   // Вернет 1
```

См. также

`RegExp`, `RegExp.exec()`, `RegExp.test()`, `String.match()`, `String.replace()`; глава 10

String.slice()

извлечение подстроки

Синтаксис

строка.`slice(`*начало*, *конец*`)`

Аргументы

начало	Индекс в строке, с которого должен начинаться фрагмент. Если этот аргумент отрицателен, он обозначает позицию, измеряемую от конца строки. То есть -1 соответствует последнему символу, -2 – второму с конца и т. д.
конец	Индекс символа исходной строки непосредственно после конца извлекаемого фрагмента. Если он не указан, фрагмент включает все символы от позиции *начало* до конца строки. Если этот аргумент отрицателен, он обозначает позицию, отсчитываемую от конца строки.

Возвращаемое значение

Новая строка, которая содержит все символы *строки*, начиная с символа в позиции *начало* (и включая его) и заканчивая символом в позиции *конец* (но не включая его).

Описание

Метод `slice()` возвращает строку, содержащую фрагмент, или подстроку *строки*, но не изменяет *строку*.

Методы `slice()`, `substring()` и признанный устаревшим метод `substr()` объекта `String` возвращают части строки. Метод `slice()` более гибок, чем `substring()`, поскольку допускает отрицательные значения аргументов. Метод `slice()` отличается от `substr()` тем, что задает подстроку с помощью двух символьных позиций, а `substr()` использует одно значение позиции и длину. Кроме того, `String.slice()` является аналогом `Array.slice()`.

Пример

```
var s = "abcdefg";
s.slice(0,4)   // Вернет "abcd"
s.slice(2,4)   // Вернет  "cd"
s.slice(4)     // Вернет  "efg"
s.slice(3,-1)  // Вернет  "def"
s.slice(3,-2)  // Вернет  "de"
s.slice(-3,-1) // Должен вернуть "ef"; в IE 4 возвращает "abcdef"
```

Ошибки

Отрицательные значения в аргументе *начало* не работают в Internet Explorer 4 (в более поздних версиях Internet Explorer эта ошибка исправлена). Они обозначают не символьную позицию, отсчитываемую от конца строки, а позицию 0.

См. также

`Array.slice()`, `String.substring()`

String.split()

разбивает строку на массив строк

Синтаксис

строка.`split`(*разделитель*, *лимит*)

Аргументы

разделитель	Строка или регулярное выражение, по которому разбивается *строка*.
лимит	Необязательное целое, определяющее максимальную длину полученного массива. Если этот аргумент указан, то количество возвращенных подстрок не будет превышать указанное. Если он не указан, разбивается вся строка независимо от ее длины.

Возвращаемое значение

Массив строк, создаваемый путем разбиения *строки* на подстроки по границам, заданным *разделителем*. Подстроки в возвращаемом массиве не включают сам разделитель, кроме описываемого далее случая.

Описание

Метод `split()` создает и возвращает массив подстрок указанной *строки*, причем размер возвращаемого массива не превышает указанный *лимит*. Эти подстроки создаются путем поиска текста, соответствующего *разделителю*, в *строке* от начала до конца и разбиения *строки* до и после найденного текста. Ограничивающий текст не включается ни в одну из возвращаемых строк, кроме случая, описываемого далее. Следует отметить, что, если разделитель соответствует началу строки, первый элемент возвращаемого массива будет пустой строкой – текстом, присутствующим перед разделителем. Аналогично, если разделитель соответствует концу строки, последний элемент массива (если это не противоречит значению аргумента *лимит*) будет пустой строкой.

Если *разделитель* не указан, *строка* вообще не разбивается и возвращаемый массив содержит только один строковый элемент, представляющий собой строку целиком. Если *разделитель* представляет собой пустую строку или регулярное выражение, соответствующее пустой строке, то *строка* разбивается между каждыми двумя соседними символами, а возвращаемый массив имеет ту же длину, что и исходная строка, если не указан меньший *лимит*. (Это особый случай, поскольку пустая строка перед первым и за последним символами отсутствует.)

Ранее отмечалось, что подстроки в массиве, возвращаемом описываемым методом, не содержат текст разделителя, использованного для разбиения строки. Однако если *разделитель* – это регулярное выражение, содержащее подвыражения в скобках, то подстроки, соответствующие этим подвыражениям (но не текст, соответствующий регулярному выражению в целом), включаются в возвращаемый массив.

Метод `String.split()` является обратным методу `Array.join()`.

Пример

Метод `split()` наиболее полезен при работе с сильно структурированными строками:

```
"1:2:3:4:5".split(":"); // Вернет ["1","2","3","4","5"]
"|a|b|c|".split("|");   // Вернет ["", "a", "b", "c", ""]
```

Еще одно распространенное применение метода `split()` состоит в разбиении команд и других подобных строк на слова, разделенные пробелами:

```
var words = sentence.split(' ');
```

Проще разбить строку на слова, используя в качестве разделителя регулярное выражение:

```
var words = sentence.split(/\s+/);
```

Чтобы разбить строку на массив символов, возьмите в качестве разделителя пустую строку. Если требуется разбить на массив символов только часть строки, задайте аргумент *лимит*:

```
"hello".split(""); // Возвращает ["h","e","l","l","o"]
"hello".split("", 3); // Возвращает ["h","e","l"]
```

Если необходимо, чтобы разделители или одна и более частей разделителя были включены в результирующий массив, напишите регулярное выражение с подвыражениями в скобках. Так, следующий код разбивает строку по HTML-тегам и включает эти теги в результирующий массив:

```
var text = "hello <b>world</b>";
text.split(/(<[^>]*>)/); // Вернет ["hello ","<b>","world","</b>",""]
```

См. также

`Array.join()`, `RegExp`; глава 10

String.substr() **устарел**

извлекает подстроку

Синтаксис

строка.`substr`(*начало*, *длина*)

Аргументы

начало Начальная позиция подстроки. Если аргумент отрицателен, он обозначает позицию, измеряемую от конца строки: -1 обозначает последний символ, -2 – второй символ с конца и т. д.

длина Количество символов в подстроке. Если этот аргумент отсутствует, возвращаемая строка включает все символы от начальной позиции до конца строки.

Возвращаемое значение

Копия фрагмента строки, начиная с символа, находящегося в позиции *начало* (включительно); имеет длину, равную аргументу *длина*, или заканчивается концом строки, если *длина* не указана.

Описание

Метод `substr()` извлекает и возвращает подстроку строки, но не изменяет строку.

Обратите внимание: метод `substr()` задает нужную подстроку с помощью позиции символа и длины. Благодаря этому появляется удобная альтернатива методам `String.substring()` и `String.splice()`, в которых подстрока задается двумя символьными позициями. При этом следует отметить, что метод не стандартизован в ECMAScript и, следовательно, считается устаревшим.

Пример

```
var s = "abcdefg";
s.substr(2,2);  // Вернет "cd"
s.substr(3);    // Вернет "defg"
s.substr(-3,2); // Должен вернуть "ef"; в IE 4 возвращает "ab"
```

Ошибки

Отрицательные значения аргумента *начало* не работают в IE 4 (в более поздних версиях IE эта ошибка исправлена). Они задают не позицию символа, отсчитываемую от конца строки, а позицию 0.

См. также

`String.slice()`, `String.substring()`

String.substring()

возвращает подстроку строки

Синтаксис

строка.`substring`(*от*, *до*)

Аргументы

от Целое, задающее позицию первого символа требуемой подстроки внутри *строки*.

до	Необязательное целое, на единицу большее позиции последнего символа требуемой подстроки. Если этот аргумент опущен, возвращаемая подстрока заканчивается в конце строки.

Возвращаемое значение

Новая строка длиной *до - от*, содержащая подстроку строки. Новая строка содержит символы, скопированные из *строки*, начиная с позиции *от* и заканчивая позицией *до*-1.

Описание

Метод `String.substring()` возвращает подстроку строки, содержащую символы между позициями *от* и *до*. Символ в позиции *от* включается в подстроку, а символ в позиции *до* не включается.

Если *от* равно *до*, метод возвращает пустую строку (длиной 0). Если *от* больше *до*, метод сначала меняет два аргумента местами, а затем возвращает строку между ними.

Помните, что символ в позиции *от* включается в подстроку, а символ в позиции *до* в нее не включается. Может показаться, что это поведение взято «с потолка» и нелогично, но особенность такой системы состоит в том, что длина возвращаемой подстроки всегда равна *до-от*.

Обратите внимание: для извлечения подстрок из строки также могут использоваться метод `String.slice()` и нестандартный метод `String.substr()`. В отличие от этих методов, `String.substring()` не может принимать отрицательные значения аргументов.

См. также

`String.charAt()`, `String.indexOf()`, `String.lastIndexOf()`, `String.slice()`, `String.substr()`

String.toLocaleLowerCase()

преобразует символы строки в нижний регистр

Синтаксис

строка.`toLocaleLowerCase()`

Возвращаемое значение

Копия *строки*, преобразованная в нижний регистр с учетом региональных параметров. Только немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что и метод `toLowerCase()`.

См. также

`String.toLocaleUpperCase()`, `String.toLowerCase()`, `String.toUpperCase()`

String.toLocaleUpperCase()

преобразует символы строки в верхний регистр

Синтаксис

строка.`toLocaleUpperCase()`

Возвращаемое значение

Копия *строки*, преобразованная в верхний регистр с учетом региональных параметров. Лишь немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что и метод `toUpperCase()`.

См. также

`String.toLocaleLowerCase()`, `String.toLowerCase()`, `String.toUpperCase()`

String.toLowerCase()

преобразует символы строки в нижний регистр

Синтаксис

строка.`toLowerCase()`

Возвращаемое значение

Копия *строки*, в которой все символы верхнего регистра преобразованы в эквивалентные им символы нижнего регистра, если такие имеются.

String.toString()

возвращает строку **переопределяет Object.toString()**

Синтаксис

строка.`toString()`

Возвращаемое значение

Элементарное строковое значение *строки*. Вызов этого метода требуется редко.

Исключения

`TypeError` Генерируется, если метод вызывается для объекта, не являющегося объектом `String`.

См. также

`String.valueOf()`

String.toUpperCase()

преобразует символы строки в верхний регистр

Синтаксис

строка.`toUpperCase()`

Возвращаемое значение

Копия *строки*, в которой все символы нижнего регистра преобразованы в эквивалентные им символы верхнего регистра, если такие имеются.

String.trim() **ECMAScript 5**

удаляет начальные и конечные пробельные символы

Синтаксис

строка.`trim()`

Возвращаемое значение

Копия *строки*, из которой удалены все начальные и конечные пробельные символы.

См. также

`String.replace()`

String.valueOf()

возвращает строку **переопределяет Object.valueOf()**

Синтаксис

строка.`valueOf()`

Возвращаемое значение

Элементарное строковое значение строки.

Исключения

`TypeError`	Генерируется, если метод вызывается для объекта, не являющегося объектом `String`.

См. также

`String.toString()`

SyntaxError

свидетельствует о синтаксической ошибке **Object→Error→SyntaxError**

Конструктор

`new SyntaxError()`

`new SyntaxError(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта `SyntaxError`.

Возвращаемое значение

Вновь созданный объект `SyntaxError`. Если указан аргумент *сообщение*, объект `SyntaxError` берет его в качестве значения своего свойства `message`; в противном случае в качестве значения этого свойства он берет строку по умолчанию, определенную в реализации. Конструктор `SyntaxError()`, вызванный как функция (без оператора `new`), ведет себя так же, как если бы он был вызван с оператором `new`.

Свойства

`message`	Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробности см. в справочной статье `Error.message`.
`name`	Строка, определяющая тип исключения. Все объекты `SyntaxError` наследуют для этого свойства строку «SyntaxError».

Описание

Экземпляр класса `SyntaxError` сигнализирует о синтаксической ошибке в программном коде. Метод `eval()`, а также конструкторы `Function()` и `RegExp()` могут генерировать исключения этого типа. Подробности о генерации и перехвате исключений см. в справочной статье `Error`.

См. также

`Error`, `Error.message`, `Error.name`

TypeError

генерируется, когда значение имеет неверный тип **Object→Error→TypeError**

Конструктор

`new TypeError()`

`new TypeError(`*сообщение*`)`

Аргументы

сообщение	Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта `TypeError`.

Возвращаемое значение

Вновь созданный объект `TypeError`. Если указан аргумент *сообщение*, объект `TypeError` берет его в качестве значения своего свойства `message`; в противном случае в качестве значения этого свойства он берет строку по умолчанию, определенную в реализации. Конструктор `TypeError()`, вызванный как функция (без оператора `new`), ведет себя так же, как если бы он был вызван с оператором `new`.

Свойства

`message`	Сообщение об ошибке, содержащее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробности см. в справочной статье `Error.message`.
`name`	Строка, определяющая тип исключения. Все объекты `TypeError` наследуют для этого свойства строку «TypeError».

Описание

Экземпляр класса `TypeError` создается, когда значение имеет не тот тип, который ожидается. Такое чаще всего происходит при попытке обратиться к свойству `null` или к неопределенному значению объекта. Это исключение может также возникнуть, если вызван метод, определенный одним классом, для объекта, являющегося экземпляром какого-либо другого класса, или если оператору `new` передается значение, не являющееся функцией-конструктором. Реализациям JavaScript также разрешено создавать объекты `TypeError`, когда встроенная функция или метод вызывается с большим числом аргументов, чем ожидается. Генерация и перехват исключений подробно рассмотрены в справочной статье `Error`.

См. также

`Error`, `Error.message`, `Error.name`

undefined

неопределенное значение

Синтаксис

`undefined`

Описание

`undefined` – это глобальное свойство, хранящее значение `undefined`. Это то же самое значение, которое возвращается при попытке прочитать значение несуществующего свойства объекта. Свойство `undefined` не перечисляется циклами `for/in` и не может быть удалено оператором `delete`. Однако `undefined` не является константой и может быть установлено равным любому другому значению, но лучше этого не делать.

Чтобы проверить, является ли значение неопределенным (`undefined`), следует использовать оператор `===`, поскольку оператор `==` считает значение `undefined` равным значению `null`.

unescape() устарело

декодирует строку с управляющими последовательностями

Синтаксис

unescape(*s*)

Аргументы

s	Декодируемая строка.

Возвращаемое значение

Декодированная копия *s*.

Описание

`unescape()` – это глобальная функция, декодирующая строку, закодированную с помощью функции `escape()`. Декодирование строки *s* происходит путем поиска и замены последовательности символов в формате %*xx* и %u*xxxx* (где *x* – шестнадцатеричная цифра) символами Юникода \u00*xx* и \u*xxxx*.

Несмотря на то что функция `unescape()` была стандартизована в первой версии ECMAScript, она признана устаревшей и исключена из стандарта в спецификации ECMAScript v3. Реализации ECMAScript могут поддерживать эту функцию, но это необязательное требование. Вместо нее следует использовать `decodeURI()` и `decodeURIComponent()`. Подробности и пример см. в справочной статье `escape()`.

См. также

`decodeURI()`, `decodeURIComponent()`, `escape()`, `String`

URIError

генерируется методами кодирования и декодирования URI **Object→Error→URIError**

Конструктор

`new URIError()`

new URIError(*сообщение*)

Аргументы

сообщение	Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта `URIError`.

Возвращаемое значение

Вновь созданный объект `URIError`. Если указан аргумент *сообщение*, объект `URIError` берет его в качестве значения своего свойства `message`; в противном случае в качестве значения этого свойства он берет строку по умолчанию, определенную в реализации. Конструктор `URIError()`, вызванный как функция (без оператора `new`), ведет себя так же, как если бы он был вызван с оператором `new`.

Свойства

`message`	Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробности см. в справочной статье `Error.message`.
`name`	Строка, определяющая тип исключения. Все объекты `URIError` наследуют для этого свойства строку «URIError».

Описание

Экземпляр класса `URIError` создается функциями `decodeURI()` и `decodeURIComponent()`, если указанная строка содержит недопустимые шестнадцатеричные управляющие последовательности. Это исключение может генерироваться методами `encodeURI()` и `encodeURIComponent()`, если указанная строка содержит недопустимые суррогатные пары символов Юникода. Генерация и перехват исключений подробно рассмотрены в справочной статье `Error`.

См. также

`Error`, `Error.message`, `Error.name`

IV

Справочник по клиентскому JavaScript

Эта часть книги представляет собой справочник по клиентскому JavaScript. Он включает описание наиболее важных объектов клиентского JavaScript, таких как `Window`, `Document`, `Element`, `Event`, `XMLHttpRequest`, `Storage`, `Canvas` и `File`. Он также содержит описание компонентов библиотеки jQuery. Справочные статьи расположены в алфавитном порядке, по именам объектов, и каждая статья включает полный список констант, свойств, методов и обработчиков событий, поддерживаемых тем или иным объектом.

В предыдущие изданиях этой книги для каждого метода отводилась отдельная справочная статья, но в этом издании справочный материал скомпонован иначе (не в ущерб детальности описания), и описание методов включено непосредственно в справочные статьи родительских объектов.

ApplicationCache
ArrayBuffer
ArrayBufferView
Attr
Audio
BeforeUnloadEvent
Blob
BlobBuilder
Button
Canvas
CanvasGradient
CanvasPattern
CanvasRenderingContext2D
ClientRect
CloseEvent
Comment
Console
ConsoleCommandLine
CSSRule
CSSStyleDeclaration
CSSStyleSheet
DataTransfer
DataView
Document
DocumentFragment
DocumentType
DOMException
DOMImplementation
DOMSettableTokenList
DOMTokenList
Element
ErrorEvent
Event
EventSource
EventTarget
FieldSet
File
FileError
FileReader
FileReaderSync
Form
FormControl
FormData
FormValidity
Geocoordinates
Geolocation
GeolocationError
Geoposition
HashChangeEvent
History
HTMLCollection
HTMLFormControlsCollection
HTMLOptionsCollection
IFrame
Image
ImageData
Input
jQuery
Label
Link
Location
MediaElement
MediaError
MessageChannel
MessageEvent
MessagePort
Meter
Navigator
Node
NodeList
Option
Output
PageTransitionEvent
PopStateEvent
ProcessingInstruction
Progress
ProgressEvent
Screen
Script
Select
Storage
StorageEvent
Style
Table
TableCell
TableRow
TableSection
Text
TextArea
TextMetrics
TimeRanges
TypedArray
URL
Video
WebSocket
Window
Worker
WorkerGlobalScope
WorkerLocation
WorkerNavigator
XMLHttpRequest
XMLHttpRequestUpload

Справочник по клиентскому JavaScript

ApplicationCache

прикладной интерфейс управления кэшем приложений **EventTarget**

Объект `ApplicationCache` – это значение свойства `applicationCache` объекта `Window`. Он определяет API управления обновлением кэшированных приложений. В простых кэшируемых приложениях не требуется использовать этот API: достаточно создать (и обновлять по мере необходимости) соответствующий файл объявления кэшируемого приложения, как описывается в разделе 20.4. В более сложных приложениях, где может возникнуть потребность более активно управлять обновлениями, можно использовать свойства, методы и обработчики событий, описываемые здесь. Подробности приводятся в разделе 20.4.2.

Константы

Следующие константы представляют допустимые значения свойства `status`.

`unsigned short` **`UNCACHED = 0`**

Это приложение не имеет атрибута `manifest`: оно не кэшируется.

`unsigned short` **`IDLE = 1`**

Файл объявления проверен, данное приложение сохранено в кэше и обновлено.

`unsigned short` **`CHECKING = 2`**

В настоящее время броузер проверяет файл объявления.

`unsigned short` **`DOWNLOADING = 3`**

Броузер загружает и сохраняет в кэше файлы, перечисленные в объявлении.

`unsigned short` **`UPDATEREADY = 4`**

Была загружена и сохранена в кэше новая версия приложения.

`unsigned short` **`OBSOLETE = 5`**

Файл объявления не найден и приложение будет удалено из кэша.

Свойства

`readonly unsigned short` **`status`**

Это свойство описывает состояние кэша текущего документа. Оно может принимать одно из значений, перечисленных выше.

Методы

`void` **`swapCache()`**

Когда свойство `status` получает значение `UPDATEREADY`, броузер управляет двумя версиями кэшируемого приложения: старые версии файлов из кэша, используемые запущенным приложением, и новые загруженные версии, которые будут использоваться при следующем запуске приложения. Приложение может вызвать `swapCache()`, чтобы сообщить броузеру, что он может немедленно удалить старые версии и начать использовать новые версии файлов. Однако имейте в виду, что это может привести к конфликту версий, и гораздо безопаснее будет выполнить переход на новую версию, перезагрузив приложение вызовом метода `Location.reload()`.

`void update()`

Обычно броузер проверяет наличие новой версии файла объявления кэшируемого приложения каждый раз, когда это приложение загружается. Долгоживущие веб-приложения могут использовать этот метод, чтобы чаще проверять наличие обновлений.

Обработчики событий

В ходе проверки файла объявления и обновления кэша броузер возбуждает в объекте `ApplicationCache` целую серию событий. Для регистрации обработчиков событий можно использовать следующие свойства объекта `ApplicationCache` или методы интерфейса `EventTarget`, реализованные в объекте `ApplicationCache`. Обработчики большинства этих событий получают простой объект `Event`. Обработчики событий «progress» получают объект `ProgressEvent`, с помощью которого можно узнать, какой объем в байтах уже был загружен.

`oncached`	Возбуждается, когда приложение впервые сохраняется в кэше. Это последнее событие в последовательности.
`onchecking`	Возбуждается, когда броузер начинает проверку наличия обновленного файла объявления. Это первое событие в последовательности событий, генерируемых в кэшируемом приложении.
`ondownloading`	Возбуждается, когда броузер начинает загрузку ресурсов, перечисленных в файле объявления, т.е. либо когда приложение впервые помещается в кэш, либо когда оно обновляется. За этим событием обычно следует одно или более событий «progress».
`onerror`	Возбуждается, когда в ходе обновления кэша возникает ошибка. Это может произойти, например, когда броузер работает в автономном режиме или если приложение ссылается на несуществующий файл объявления.
`onnoupdate`	Возбуждается, когда броузер определяет, что файл объявления не изменился и приложение в кэше имеет текущую версию. Это последнее событие в последовательности.
`onobsolete`	Возбуждается, когда исчезает файл объявления кэшируемого приложения. Это приводит к удалению приложения из кэша. Это последнее событие в последовательности.
`onprogress`	Возбуждается периодически, пока идет загрузка и сохранение в кэше файлов приложения. С этим событием обработчикам передается объект `ProgressEvent`.
`onupdateready`	Возбуждается, когда броузер загрузит и сохранит в кэше новую версию приложения (и она будет готова к использованию при следующем запуске приложения). Это последнее событие в последовательности.

ArrayBuffer

последовательность байтов фиксированной длины

Объект `ArrayBuffer` представляет последовательность байтов фиксированной длины в памяти, но не определяет методов извлечения и сохранения байтов. Доступ к байтам и их интерпретацию обеспечивает объект `ArrayBufferView`, подобный классам типизированных массивов.

Конструктор

`new` **`ArrayBuffer`**`(unsigned long` *`length`*`)`

Создает новый объект `ArrayBuffer` с указанным количеством байтов. Все байты в новом объекте `ArrayBuffer` инициализируются значением 0.

Свойства

`readonly unsigned long` **`byteLength`**

Длина в байтах последовательности в объекте `ArrayBuffer`.

ArrayBufferView

общие свойства типов, основанных на ArrayBuffer

Тип `ArrayBufferView` служит суперклассом для классов, предоставляющих доступ к байтам в объекте `ArrayBuffer`. Объект `ArrayBufferView` нельзя создать непосредственно: он предназначен, чтобы определять общие свойства для подтипов, таких как типизированные массивы и `DataView`.

Свойства

`readonly ArrayBuffer` **`buffer`**

Объект `ArrayBuffer`, представлением которого является данный объект.

`readonly unsigned long` **`byteLength`**

Длина в байтах фрагмента буфера, доступного посредством данного представления.

`readonly unsigned long` **`byteOffset`**

Начальная позиция в байтах фрагмента буфера, доступного посредством данного представления.

Attr

атрибут элемента документа

Объект `Attr` представляет атрибут узла `Element`. Получить объект `Attr` можно посредством свойства `attributes` интерфейса `Node` или вызовом метода `getAttributeNode()` или `getAttributeNodeNS()` интерфейса `Element`.

Поскольку значения атрибутов могут быть представлены в виде строк, обычно нет необходимости использовать интерфейс `Attr`. В большинстве случаев самый простой способ работы с атрибутами предоставляют методы `Element.getAttribute()` и `Element.setAttribute()`. Эти методы используют строки в качестве значений атрибутов и позволяют вообще отказаться от применения объектов `Attr`.

Свойства

`readonly string` **`localName`**

Имя атрибута, без возможного префикса пространства имен.

`readonly string` **name**

Имя атрибута, включая префикс пространства имен, если таковое имеется.

`readonly string` **namespaceURI**

Идентификатор URI, определяющий пространство имен атрибута, или `null`, если отсутствует.

`readonly string` **prefix**

Префикс пространства имен атрибута или `null`, если отсутствует.

`string` **value**

Значение атрибута.

Audio

HTML-элемент <audio> **Node, Element, MediaElement**

Объект `Audio`, представляющий HTML-элемент `<audio>`. Объект `Audio` не имеет свойств, методов и обработчиков событий, кроме конструктора, помимо тех, что унаследованы от `MediaElement`.

Конструктор

`new` **Audio**(`[string` *src*`]`)

Этот конструктор создает новый элемент `<audio>` с атрибутом `preload`, установленным в значение «auto». Если указан аргумент *src*, он используется как значение атрибута `src`.

BeforeUnloadEvent

объект Event для событий выгрузки **Event**

Событие «unload» возбуждается в объекте `Window` непосредственно перед тем, как броузер перейдет к другому документу; оно дает веб-приложению возможность предложить пользователю подтвердить свое желание покинуть страницу. Обработчикам события «unload» передается объект `BeforeUnloadEvent`. Если вам потребуется запросить у пользователя подтвердить желание покинуть страницу, вам не нужно вызывать метод `Window.confirm()`. Вместо этого верните из обработчика события строку или присвойте строку свойству `returnValue` этого объекта. Эта строка будет выведена перед пользователем в форме диалога подтверждения.

Смотрите также справочные статьи `Event` и `Window`.

Свойства

`string` **returnValue**

Сообщение, отображаемое в диалоге подтверждения перед уходом с текущей страницы. Оставьте это свойство пустым, если вам не требуется отображать диалог подтверждения.

Blob

блок двоичных данных, таких как содержимое файла

Объекты `Blob` используются для организации обмена данными между различными прикладными интерфейсами. Объекты `Blob` могут иметь очень большой размер и способны представлять блоки двоичных данных, но ни то, ни другое не является обязательным. Объекты `Blob` часто сохраняются в файлах, но это зависит от реализации.

Объекты `Blob` позволяют узнать только свой размер и иногда MIME-тип хранящихся в них данных и определяют единственный метод, позволяющий интерпретировать фрагмент своих данных как отдельный объект `Blob`.

Объекты `Blob` используются многими прикладными интерфейсами: объект `FileReader` позволяет читать содержимое объекта `Blob`, а объект `BlobBuilder` – создавать новые объекты `Blob`. Объект `XMLHttpRequest` обеспечивает возможность загружать и выгружать объекты `Blob`. Обсуждение объектов `Blob` и прикладных интерфейсов, использующих их, вы найдете в разделе 22.6.

Свойства

readonly unsigned long **size**

Объем двоичных данных в объекте `Blob` в байтах.

readonly string **type**

MIME-тип данных в объекте `Blob`, если указан, в противном случае – пустая строка.

Методы

Blob **slice**(unsigned long *start*, unsigned long *length*, [string *contentType*])

Возвращает новый объект `Blob`, представляющий *length* байтов в данном объекте `Blob`, начиная со смещения *start*. Если указан аргумент *contentType*, он будет использован, как значение свойства `type` возвращаемого объекта `Blob`

BlobBuilder

создает новые объекты Blob

Объект `BlobBuilder` используется для создания новых объектов `Blob` из текстовых строк и из двоичных данных в объектах `ArrayBuffer` и в других объектах `Blob`. Чтобы создать объект `Blob`, следует сначала создать объект `BlobBuilder`, вызвать его метод `append()` один или более раз и затем вызвать метод `getBlob()`.

Конструктор

new **BlobBuilder()**

Новый объект `BlobBuilder` создается вызовом конструктора `BlobBuilder()` без аргументов.

Методы

void **append**(string *text*, [string *endings*])

Добавляет в конструируемый двоичный объект `Blob` текст *text* в кодировке UTF-8.

void **append**(Blob *data*)

Добавляет в конструируемый двоичный объект `Blob` данные из двоичного объекта *data*.

void **append**(ArrayBuffer *data*)

Добавляет в конструируемый двоичный объект `Blob` данные из объекта *data* типа `ArrayBuffer`.

Blob **getBlob**([string *contentType*])

Возвращает объект `Blob`, представляющий все данные, которые были добавлены в этот объект `BlobBuilder` с момента его создания. Каждый вызов этого метода возвращает новый объект `Blob`. Если указан аргумент *contentType*, он будет использоваться в качестве значения свойства `type` возвращаемого объекта `Blob`. Если этот аргумент не указан, свойство `type` возвращаемого объекта `Blob` будет содержать пустую строку.

Button

HTML-элемент <button> **Node, Element, FormControl**

Объект `Button` представляет HTML-элемент `<button>`. Большинство свойств и методов объекта `Button` описываются в справочных статьях `FormControl` и `Element`. Однако, когда свойство `type` объекта `Button` (смотрите справочную статью `FormControl`) имеет значение «submit», другие свойства, перечисленные здесь, определяют параметры отправки формы, имеющие приоритет перед аналогичными свойствами формы, в которой находится кнопка `Button` (смотрите справочную статью `FormControl`).

Свойства

Следующие свойства используются, только когда элемент `<button>` имеет атрибут `type` со значением «submit».

`string` **`formAction`**

Это свойство соответствует HTML-атрибуту `formaction`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `action` форм.

`string` **`formEnctype`**

Это свойство соответствует HTML-атрибуту `formenctype`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `enctype` форм и может принимать те же значения.

`string` **`formMethod`**

Это свойство соответствует HTML-атрибуту `formmethod`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `method` форм.

`string` **`formNoValidate`**

Это свойство соответствует HTML-атрибуту `formnovalidate`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `noValidate` форм.

`string` **`formTarget`**

Это свойство соответствует HTML-атрибуту `formtarget`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `target` форм.

Canvas

HTML-элемент для создания графических изображений **Node, Element**

Объект Canvas представляет HTML-элемент `<canvas>`. Он не обладает собственным поведением, но определяет API для поддержки операций рисования. С помощью этого объекта можно задать ширину и высоту холста с помощью его свойств `width` и `height`, а вызовом метода `toDataUrl()` из него можно извлечь изображение, но основная функциональность обеспечивается объектом «контекста», возвращаемого методом `getContext()`. Смотрите справочную статью `CanvasRenderingContext2D`.

Свойства

`unsigned long` **`height`**

`unsigned long` **`width`**

Эти свойства соответствуют атрибутам `width` и `height` тега `<canvas>` и определяют размеры координатной плоскости холста. По умолчанию свойство `width` имеет значение 300, а `height` – 150.

Объекты `Blob` позволяют узнать только свой размер и иногда MIME-тип хранящихся в них данных и определяют единственный метод, позволяющий интерпретировать фрагмент своих данных как отдельный объект `Blob`.

Объекты `Blob` используются многими прикладными интерфейсами: объект `FileReader` позволяет читать содержимое объекта `Blob`, а объект `BlobBuilder` – создавать новые объекты `Blob`. Объект `XMLHttpRequest` обеспечивает возможность загружать и выгружать объекты `Blob`. Обсуждение объектов `Blob` и прикладных интерфейсов, использующих их, вы найдете в разделе 22.6.

Свойства

`readonly unsigned long` **`size`**

Объем двоичных данных в объекте `Blob` в байтах.

`readonly string` **`type`**

MIME-тип данных в объекте `Blob`, если указан, в противном случае – пустая строка.

Методы

`Blob` **`slice`**`(unsigned long` *`start`*`, unsigned long` *`length`*`, [string` *`contentType`*`])`

Возвращает новый объект `Blob`, представляющий *`length`* байтов в данном объекте `Blob`, начиная со смещения *`start`*. Если указан аргумент *`contentType`*, он будет использован, как значение свойства `type` возвращаемого объекта `Blob`

BlobBuilder

создает новые объекты Blob

Объект `BlobBuilder` используется для создания новых объектов `Blob` из текстовых строк и из двоичных данных в объектах `ArrayBuffer` и в других объектах `Blob`. Чтобы создать объект `Blob`, следует сначала создать объект `BlobBuilder`, вызвать его метод `append()` один или более раз и затем вызвать метод `getBlob()`.

Конструктор

`new` **`BlobBuilder()`**

Новый объект `BlobBuilder` создается вызовом конструктора `BlobBuilder()` без аргументов.

Методы

`void` **`append`**`(string` *`text`*`, [string` *`endings`*`])`

Добавляет в конструируемый двоичный объект `Blob` текст *`text`* в кодировке UTF-8.

`void` **`append`**`(Blob` *`data`*`)`

Добавляет в конструируемый двоичный объект `Blob` данные из двоичного объекта *`data`*.

`void` **`append`**`(ArrayBuffer` *`data`*`)`

Добавляет в конструируемый двоичный объект `Blob` данные из объекта *`data`* типа `ArrayBuffer`.

`Blob` **`getBlob`**`([string` *`contentType`*`])`

Возвращает объект `Blob`, представляющий все данные, которые были добавлены в этот объект `BlobBuilder` с момента его создания. Каждый вызов этого метода возвращает новый объект `Blob`. Если указан аргумент *`contentType`*, он будет использоваться в качестве значения свойства `type` возвращаемого объекта `Blob`. Если этот аргумент не указан, свойство `type` возвращаемого объекта `Blob` будет содержать пустую строку.

Button

HTML-элемент <button> **Node, Element, FormControl**

Объект `Button` представляет HTML-элемент `<button>`. Большинство свойств и методов объекта `Button` описываются в справочных статьях `FormControl` и `Element`. Однако, когда свойство `type` объекта `Button` (смотрите справочную статью `FormControl`) имеет значение «submit», другие свойства, перечисленные здесь, определяют параметры отправки формы, имеющие приоритет перед аналогичными свойствами формы, в которой находится кнопка `Button` (смотрите справочную статью `FormControl`).

Свойства

Следующие свойства используются, только когда элемент `<button>` имеет атрибут `type` со значением «submit».

`string` **`formAction`**

Это свойство соответствует HTML-атрибуту `formaction`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `action` форм.

`string` **`formEnctype`**

Это свойство соответствует HTML-атрибуту `formenctype`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `enctype` форм и может принимать те же значения.

`string` **`formMethod`**

Это свойство соответствует HTML-атрибуту `formmethod`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `method` форм.

`string` **`formNoValidate`**

Это свойство соответствует HTML-атрибуту `formnovalidate`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `noValidate` форм.

`string` **`formTarget`**

Это свойство соответствует HTML-атрибуту `formtarget`. Для кнопок, управляющих отправкой форм, это свойство переопределяет свойство `target` форм.

Canvas

HTML-элемент для создания графических изображений **Node, Element**

Объект Canvas представляет HTML-элемент `<canvas>`. Он не обладает собственным поведением, но определяет API для поддержки операций рисования. С помощью этого объекта можно задать ширину и высоту холста с помощью его свойств `width` и `height`, а вызовом метода `toDataUrl()` из него можно извлечь изображение, но основная функциональность обеспечивается объектом «контекста», возвращаемого методом `getContext()`. Смотрите справочную статью `CanvasRenderingContext2D`.

Свойства

`unsigned long` **`height`**

`unsigned long` **`width`**

Эти свойства соответствуют атрибутам `width` и `height` тега `<canvas>` и определяют размеры координатной плоскости холста. По умолчанию свойство `width` имеет значение 300, а `height` – 150.

Если размер элемента `<canvas>` не указан в таблице стилей или во встроенном атрибуте `style`, эти свойства `width` и `height` также определяют экранные размеры холста.

Изменение значений этих свойств (даже запись их текущих значений) вызывает очистку холста (заливку черным прозрачным цветом) и сброс всех его графических атрибутов в значения по умолчанию.

Методы

object **getContext**(string *contextId*, [любые *аргументы*...])

Возвращает объект, посредством которого выполняется рисование в элементе Canvas. Если передать ему строку «2d», он вернет объект CanvasRenderingContext2D, реализующий рисование на двухмерной плоскости. В этом случае не требуется передавать никаких дополнительных аргументов.

Для каждого элемента `<canvas>` существует только один объект CanvasRenderingContext2D, т.е. повторные вызовы `getContext("2d")` будут возвращать тот же самый объект.

Спецификация HTML5 стандартизует аргумент «2d» для этого метода и не определяет других допустимых аргументов. В настоящее время разрабатывается отдельный стандарт, WebGL, для трехмерной графики. В броузерах, поддерживающих его, этому методу можно передать строку «webgl», чтобы получить объект, обеспечивающий создание трехмерных изображений.

Следует, однако, отметить, что в данной книге описывается только объект CanvasRenderingContext2D.

string **toDataURL**([string *type*], [любые *аргументы*...])

Метод `toDataURL()` возвращает растровое изображение холста в виде URL-адреса `data://`, который можно использовать в теге `<img>` или передавать по сети. Например:

```
// Скопировать содержимое холста в элемент <img> и добавить его в документ
var canvas = document.getElementById("my_canvas");
var image = document.createElement("img");
image.src = canvas.toDataURL();
document.body.appendChild(image);
```

Аргумент *type* определяет, какой MIME-тип или графический формат изображения следует использовать. Если этот аргумент отсутствует, используется значение по умолчанию «image/png». Формат PNG является единственным, который обязаны поддерживать все реализации. Чтобы получить изображение в другом формате, отличном от PNG, можно передать дополнительные аргументы, определяющие порядок кодирования. Например, если в аргументе *type* передается строка «image/jpeg», во втором аргументе следует передать число в диапазоне от 0 до 1, определяющее уровень качества изображения. На момент написания этих строк никаких других аргументов стандартизовано не было.

Для предотвращения утечки информации между документами с разным происхождением метод `toDataURL()` не будет работать с тегами `<canvas>`, которые имеют «неясное происхождение». Считается, что элемент `<canvas>` имеет неясное происхождение, если в нем содержалось изображение (созданное непосредственно с помощью метода `drawImage()` или косвенно, с помощью объекта CanvasPattern), имеющее иное происхождение, отличное от происхождения содержащего его документа. Кроме того, считается, что элемент `<canvas>` имеет неясное происхождение, если в нем рисовался текст с использованием веб-шрифтов, имеющих иное происхождение.

CanvasGradient

цветной градиент для использования в элементе Canvas

Объект `CanvasGradient` представляет цветовой градиент, который может быть присвоен свойствам `strokeStyle` и `fillStyle` объекта `CanvasRenderingContext2D`. Объект `CanvasGradient` возвращается методами `createLinearGradient()` и `createRadialGradient()` объекта `CanvasRenderingContext2D`.

После создания объекта `CanvasGradient` следует вызвать метод `addColorStop()` и с его помощью определить, какой цвет в какой позиции должен отображаться внутри градиента. Между заданными позициями цвет будет интерполироваться так, чтобы создать эффект плавного перехода или исчезновения цвета. Если не определить цвет ни в одной позиции, градиент будет окрашен однородным прозрачным черным цветом.

Методы

`void` **`addColorStop`**`(double` *`offset`*`, string` *`color`*`)`

Метод `addColorStop()` определяет фиксированные цвета внутри градиента. В аргументе *`color`* передается строка с названием цвета в формате CSS. В аргументе *`offset`* передается значение с плавающей точкой в диапазоне от 0.0 до 1.0, которое представляет позицию между началом и концом градиента. Значение 0 соответствует начальной позиции, значение 1 – конечной.

Если указать два или более цвета, градиент создаст эффект плавного перехода цветов между указанными позициями. Перед первой позицией будет отображаться цвет, соответствующий первой позиции. После последней позиции градиент будет отображать цвет, соответствующий последней позиции. Если определить цвет только для одной позиции, градиент будет окрашен одним цветом. Если не определить цвет ни в одной позиции, градиент будет окрашен однородным прозрачным черным цветом.

CanvasPattern

шаблон заполнения холста на основе готового изображения

Объект `CanvasPattern` возвращается методом `createPattern()` объекта `CanvasRenderingContext2D`. Объект `CanvasPattern` может использоваться в качестве значения свойств `strokeStyle` и `fillStyle` объекта `CanvasRenderingContext2D`.

CanvasRenderingContext2D

объект, используемый для создания изображений

Объект `CanvasRenderingContext2D` предоставляет набор свойств и методов для рисования двухмерных графических изображений. Следующие разделы содержат краткий обзор его возможностей. Дополнительная информация приводится в разделе 21.4, а также в справочных статьях `Canvas`, `CanvasGradient`, `CanvasPattern`, `ImageData` и `TextMetrics`.

Создание и отображение контуров

Очень мощная особенность элемента `Canvas` заключается в возможности строить фигуры с помощью элементарных операций рисования и затем отображать их либо в виде ограничивающих фигуру линий, либо заполнять их. Собранные воедино операции рисования называются *текущим контуром*. Элемент `Canvas` поддерживает возможность работы лишь с одним текущим контуром.

Для построения связанной фигуры из отдельных сегментов для каждой промежуточной операции рисования должна быть определена точка присоединения. Для этой цели `Canvas` поддерживает *текущую позицию*. Операции рисования неявно используют ее в качестве начальной точки и обычно переустанавливают текущую позиции в свою конечную точку. Это можно представить себе как перемещение пера по листу бумаги: когда заканчивается рисование отдельной линии, текущей становится позиция, в которой было остановлено движение пера.

Существует возможность создать в текущем контуре несколько несвязанных фигур, которые должны отображаться с одними и теми же параметрами рисования. Для отделения фигур используется метод `moveTo()`; он перемещает текущую позицию в новые координаты без добавления линии, связывающей точки. Когда вызывается этот метод, создается новый вложенный контур, или *подконтур*, который в терминах элемента `Canvas` используется для объединения связанных операций.

Из доступных операций рисования можно упомянуть: `lineTo()` – рисование отрезков прямых линий, `rect()` – рисование прямоугольников, `arc()` и `arcTo()` – рисование дуг, `bezierCurveTo()` и `quadraticCurveTo()` – рисование кривых.

Как только требуемый контур сформирован, нарисовать фигуру в виде ограничивающих линий можно методом `stroke()`, а залить внутреннюю область фигуры – методом `fill()`; можно также вызвать оба метода.

Помимо рисования линий и заливки фигур текущий контур можно использовать как *область отсечки*. Пикселы в пределах этой области будут отображаться, за ее пределами – нет. Область отсечки обладает свойством накапливать изменения – вызов метода `clip()` для создания области отсечки, пересекающейся с текущей, приводит к созданию новой объединенной области.

Если сегменты в любом из вложенных контуров не формируют замкнутую фигуру, операции `fill()` и `clip()` неявно замыкают их, добавляя виртуальный (невидимый) отрезок прямой линии, соединяющий начальную и конечную точки контура. Чтобы явно добавить такой сегмент и тем самым замкнуть фигуру, следует вызвать метод `closePath()`.

Чтобы проверить, находится ли точка внутри (или на границе) текущего контура, можно использовать метод `isPointInPath()`. Когда контур пересекает сам себя или состоит из нескольких накладывающихся друг на друга подконтуров, понятие «внутри» определяется правилом сохранения знака. Если нарисовать одну окружность внутри другой и обе рисовать в одном и том же направлении, все, что находится внутри большей окружности, будет считаться внутренней областью контура. Если, напротив, одну окружность нарисовать по часовой стрелке, а другую – против часовой стрелки, получится кольцо, и внутренняя область меньшей окружности будет считаться за пределами контура. Это же определение внутренней области используется методами `fill()` и `clip()`.

Цвета, градиенты и шаблоны

При заполнении или рисовании границ фигуры существует возможность указать, каким образом должны заполняться линии или ограниченная ими область, для чего используются свойства `fillStyle` и `strokeStyle`. Оба эти свойства могут принимать строку со значением цвета в формате CSS, а также объект `CanvasGradient` или `CanvasPattern`, который описывает градиент или шаблон. Для создания градиента используется метод `createLinearGradient()` или `createRadialGradient()`, для создания шаблона – метод `createPattern()`.

Непрозрачный цвет в нотации CSS задается строкой в формате «#RRGGBB», где RR, GG и BB – это шестнадцатеричные цифры, определяющие интенсивность красной (red), зеленой (green) и синей (blue) составляющих в диапазоне от 00 до FF. Например, ярко-красный цвет описывается строкой «#FF0000». Чтобы определить степень прозрачности цвета, используется строка в формате «rgba(R, G, B, A)». В такой нотации R, G и B определяют интенсивность красной, зеленой и синей составляющих цвета в виде десятичных чисел в диапазоне от 0 до 255, а A – альфа-компонент (прозрачность), как число с плавающей точкой в диапазоне от 0.0 (полностью прозрачный цвет) до 1.0 (полностью непрозрачный цвет). Например, полупрозрачный ярко-красный цвет описывается строкой «rgba(255, 0, 0, 0.5)».

Толщина, окончания и сопряжение линий

Элемент `Canvas` имеет несколько свойств, с помощью которых можно определить различные варианты отображения линий. Толщину линий можно указать с помощью свойства `lineWidth`, окончания линий – с помощью свойства `lineCap`, сопряжения линий – с помощью свойства `lineJoin`.

Рисование прямоугольников

Нарисовать и заполнить прямоугольник можно с помощью методов `strokeRect()` и `fillRect()`. Кроме того, методом `clearRect()` можно очистить прямоугольную область.

Рисование изображений

В API объекта `Canvas` изображения определяются с помощью объектов `Image`, которые представляют HTML-теги `<img>` или невидимые изображения, созданные с помощью конструктора `Image()` (дополнительную информацию см. в справочной статье `Image`). Кроме того, в качестве объекта-источника изображения могут использоваться элементы `<canvas>` и `<video>`.

Добавить изображение в элемент `Canvas` можно с помощью метода `drawImage()`, который в наиболее общем случае позволяет масштабировать и выводить на экран произвольный прямоугольный участок исходного изображения.

Рисование текста

Метод `fillText()` рисует текст, а метод `strokeText()` рисует контуры букв, составляющих текст. Используемый шрифт определяется свойством `font`; значение этого свойства должно быть строкой определения шрифта в формате CSS. Выравнивание текста относительно указанной координаты X по левому краю, по правому краю или по центру определяется с помощью свойства `textAlign`, а выравнивание относительно указанной координаты Y – с помощью свойства `textBaseline`.

Система координат и преобразования

По умолчанию начало системы координат холста находится в точке `(0,0)`, в верхнем левом углу, когда координата X растет в направлении к правой границе, а координата Y – к нижней. Атрибуты `width` и `height` тега `<canvas>` определяют максимальные значения координат X и Y, а одна элементарная единица измерения в системе координат обычно соответствует одному пикселу.

Однако существует возможность преобразовать систему координат, вызывая смещение, масштабирование или вращение системы координат в операциях рисования. Делается это с помощью методов `translate()`, `scale()` и `rotate()`, которые оказывают влияние на *матрицу преобразования* холста. Поскольку система координат может подвергаться подобным преобразованиям, значения координат, которые передаются методам, таким как `lineTo()`, могут не соответствовать количеству пикселов. По этой причине для определения координат в API объекта `Canvas` используются не целые, а вещественные числа.

Тени

Объект `CanvasRenderingContext2D` может автоматически добавлять тени к любым создаваемым фигурам. Цвет тени задается с помощью свойства `shadowColor`, а ее смещение – с помощью свойств `shadowOffsetX` и `shadowOffsetY`. Кроме того, с помощью свойства `shadowBlur` можно определить степень размытия краев тени.

Композиция

Обычно при рисовании на холсте новые фигуры накладываются поверх ранее нарисованных, частично или полностью скрывая их, в зависимости от степени прозрачности новых фигур. Процесс объединения новых пикселов со старыми называется «композицией», и, указывая различные значения в свойстве `globalCompositeOperation`, можно управлять порядком объединения пикселов. Например, это свойство можно установить так, что новые фигуры будут рисоваться под существующими.

В следующей таблице перечислены допустимые значения свойства и их смысл. Под *исходными* в таблице подразумеваются пикселы, которые рисуются в настоящий момент, под *целевыми* – существующие пикселы. Под *результирующими* – пикселы, которые получаются в результате объединения исходных и целевых пикселов. В формулах символом `S` обозначается исходный (source) пиксел, символом `D` – целевой (destination) пиксел, символом `R` – результирующий (result) пиксел, символом α_s – альфа-компонент (уровень непрозрачности) исходного пиксела, и символом α_d – альфа-компонент целевого пиксела:

Значение	**Формула**	**Описание**
`"copy"`	$R = S$	Исходный пиксел рисуется без учета целевого пиксела.
`"destination-atop"`	$R = (1-\alpha_d)S + \alpha_s D$	Исходный пиксел рисуется под целевым. Если исходный пиксел является прозрачным, результирующий пиксел также будет прозрачным.
`"destination-in"`	$R = \alpha_s D$	Целевой пиксел умножается на непрозрачность исходного, но цвет исходного пиксела игнорируется.
`"destination-out"`	$R = (1-\alpha_s)D$	Целевой пиксел делается прозрачным, если исходный пиксел непрозрачен, и остается без изменений, если исходный пиксел прозрачен. Цвет исходного пиксела игнорируется.
`"destination-over"`	$R = (1-\alpha_d)S + D$	Исходный пиксел рисуется под целевым, и его видимость зависит от прозрачности целевого пиксела.
`"lighter"`	$R = S + D$	Цветовые составляющие двух пикселов просто складываются, а их суммы обрезаются, если превышают максимально возможное значение.
`"source-atop"`	$R = \alpha_d S + (1-\alpha_s)D$	Исходный пиксел рисуется поверх целевого, но умножается на его непрозрачность. Поверх совершенно прозрачного целевого пиксела ничего не рисуется.
`"source-in"`	$R = \alpha_d S$	Исходный пиксел умножается на непрозрачность целевого. Цвет целевого пиксела игнорируется. Если целевой пиксел является прозрачным, результирующий пиксел также будет прозрачным.
`"source-out"`	$R = (1-\alpha_d)S$	Результирующий пиксел получит цвет исходного, если целевой пиксел прозрачен, и прозрачным, если целевой пиксел непрозрачен. Цвет целевого пиксела игнорируется.

Значение	Формула	Описание
"source-over"	$R = S + (1-\alpha_s)D$	Исходный пиксел рисуется поверх целевого. Если исходный пиксел является полупрозрачным, цвет целевого пиксела будет влиять на цвет результата.* Это значение является значением по умолчанию свойства globalCompositeOperation.
"xor"	$R = (1-\alpha_d)S + (1-\alpha_s)D$	Если исходный пиксел прозрачен, результатом композиции станет целевой пиксел. Если целевой пиксел прозрачен, результатом станет исходный пиксел. Если оба пиксела, исходный и целевой, являются прозрачными или непрозрачными, в результате получится прозрачный пиксел.

Сохранение значений графических параметров

Методы save() и restore() позволяют сохранять и восстанавливать параметры объекта CanvasRenderingContext2D. Метод save() помещает параметры на вершину стека, а метод restore() снимает последние сохраненные параметры с вершины стека и делает их текущими.

Сохраняются все свойства объекта CanvasRenderingContext2D (за исключением свойства canvas, которое является константой). Матрица преобразования и область отсечки также сохраняются на стеке, но текущий контур и позиция пера не сохраняются.

Манипулирование пикселами

Метод getImageData() позволяет получить массив пикселов холста, а метод putImageData()дает возможность устанавливать отдельные пикселы. Эти методы могут пригодиться, если потребуется реализовать обработку изображений на языке JavaScript.

Свойства

readonly Canvas **canvas**

Элемент Canvas, который будет использоваться для создания изображения.

any **fillStyle**

Текущий цвет, шаблон или градиент, используемый для заполнения. Это свойство может принимать строковое значение либо объект CanvasGradient или CanvasPattern. По умолчанию заливка выполняется сплошным черным цветом

string **font**

Шрифт, используемый методами рисования текста. Для определения значения этого свойства используется тот же синтаксис, что и при определении значения CSS-атрибута font. Значение по умолчанию: «10px sans-serif». Если размер шрифта в строке указан в таких единицах, как «em» или «ex», или используются ключевые слова, определяющие относительные значения, такие как «larger», «smaller», «bolder» или «lighter», они интерпретируются относительно вычисленного CSS-стиля шрифта элемента <canvas>.

double **globalAlpha**

Определяет дополнительный уровень прозрачности, который будет добавляться ко всему, что будет нарисовано на холсте. Значение альфа-компонента всех пикселов, рисуемых на холсте, будет умножаться на значение этого свойства. Диапазон значений от 0.0 (полностью прозрачный) до 1.0 (значение по умолчанию: не добавляет дополнительную прозрачность).

`string` **globalCompositeOperation**

Определяет порядок смешения («композиции») цветов на холсте. Обычно это свойство бывает полезным только при работе с полупрозрачными цветами или когда изменяется значение свойства `globalAlpha`. Значение по умолчанию: «source-over». Также часто используются значения «destination-over» и «copy». Перечень допустимых значений приводится в таблице выше. Обратите внимание, что на момент написания этих строк броузеры по-разному выполняли некоторые виды композиции: некоторые выполняли композицию локально, а некоторые – глобально. Подробности приводятся в разделе 21.4.13.

`string` **lineCap**

Определяет, как должны оканчиваться отображаемые линии. Устанавливать это свойство имеет смысл только при рисовании толстых линий. Допустимые значения перечислены в следующей таблице. Значение по умолчанию: «butt».

Значение	Смысл
`"butt"`	Это значение по умолчанию. Оно указывает, что окончания линий не должны рисоваться. В этом случае конец линии выглядит просто как перпендикуляр к боковым сторонам линии. Линия не выступает за свои конечные точки.
`"round"`	Это значение указывает, что линия должна иметь наконечник в виде полукруга с диаметром, равным толщине линии; в результате линия выступает за конечные точки на половину своей толщины.
`"square"`	Это значение указывает, что линия должна иметь окончание в виде квадрата. Это значение по своему поведению напоминает значение «butt», но при использовании данного значения линия выступает за конечные точки на половину своей толщины.

`string` **lineJoin**

Когда контур включает вершины, где соединяются прямые линии и/или кривые, свойство `lineJoin` определяет, как должны рисоваться эти вершины. Действие этого свойства проявляется только при рисовании толстых линий.

Значение по умолчанию, «miter», указывает, что внешние края двух линий в точке сопряжения должны быть продолжены, пока они не пересекутся. Когда две линии сопрягаются под очень острым углом, область сопряжения может оказаться достаточно длинной. Ограничить максимальную длину такого варианта сопряжения можно с помощью свойства `miterLimit`. Когда длина сопряжения превышает этот предел, сопряжение просто усекается.

Значение «round» указывает, что внешние края линий, образующих вершину, должны сопрягаться закрашенной дугой, диаметр которой равен толщине линий. Значение «bevel» указывает, что внешние края линий, образующих вершину, должны сопрягаться закрашенным треугольником.

`double` **lineWidth**

Определяет толщину линий для операций рисования. Значение по умолчанию – 1. Широкие линии центрируются по воображаемой линии контура на половину толщины в одну сторону и на половину толщины в другую.

`double` **miterLimit**

Когда линии сопрягаются под очень острым углом и при этом свойство `lineJoin` установлено в значение «miter», область сопряжения может оказаться достаточно длинной. Слишком длинная область сопряжения может выглядеть достаточно некрасиво. Свойство `miterLimit` позволяет определить максимальную длину сопряжения. Величина этого свойства выражает отношение длины области сопряжения к толщине линий. Значение по умолчанию – 10, оно означает, что область сопряжения по длине никогда не должна превышать толщину линий более чем в 5 раз. Когда длина сопряжения превышает этот предел, оно просто усекается.

`double` **`shadowBlur`**

Определяет степень размытия краев тени. Значение по умолчанию – 0; при этом тень воспроизводится с резкими краями. Чем больше значение, тем более размытый край тени получается. Имейте в виду, что это значение измеряется не в пикселах и не подвержено действию текущего преобразования системы координат.

`string` **`shadowColor`**

Определяет цвет тени в формате CSS. По умолчанию используется черный прозрачный цвет.

`double` **`shadowOffsetX`**

`double` **`shadowOffsetY`**

Определяют горизонтальное и вертикальное смещение теней. Чем больше смещение, тем выше объект с тенью кажется расположенным над фоном. Значение по умолчанию: 0. Эти значения измеряются в единицах системы координат и не зависят от текущего преобразования.

`any` **`strokeStyle`**

Определяет цвет, шаблон или градиент, используемый для рисования контуров. Это свойство может принимать строку с обозначением цвета в формате CSS либо объект `CanvasGradient` или `CanvasPattern`.

`string` **`textAlign`**

Определяет выравнивание текста по горизонтали относительно координаты X, передаваемой методам `fillText()` и `strokeText()`. Допустимыми значениями являются «left», «center», «right», «start» и «end». Смысл значений «start» и «end» зависит от атрибута `dir` (направление письма) тега `<canvas>`. Значение по умолчанию – «start».

`string` **`textBaseline`**

Определяет выравнивание текста по вертикали относительно координаты Y, передаваемой методам `fillText()` и `strokeText()`. Допустимыми значениями являются «top», «middle», «bottom», «alphabetic», «hanging» и «ideographic». Значение по умолчанию – «alphabetic».

Методы

`void` **`arc`**`(double` *`x,y,radius,startAngle,endAngle`*`, [boolean` *`anticlockwise`*`])`

Добавляет в текущий подконтур дугу с заданными центром окружности и радиусом. Первые три аргумента этого метода описывают координаты центра и радиус окружности. Следующие два аргумента – это углы, определяющие начальную и конечную точки дуги на окружности. Углы измеряются в радианах. Позиция, соответствующая трем часам на циферблате, т.е. положительной оси X, имеет угол 0. Углы увеличиваются по направлению часовой стрелки. Последний аргумент определяет направление рисования дуги – против часовой стрелки (`true`) или по часовой стрелке (`false`).

Вызов этого метода добавляет в текущий подконтур отрезок прямой линии между текущей позицией пера и начальной точкой дуги и затем добавляет дугу.

void **arcTo**(double *x1, y1, x2, y2, radius*)

Добавляет в текущий подконтур прямую линию и дугу, описывая эту дугу таким образом, что этот метод особенно удобно использовать для рисования закругленных углов многоугольников. Аргументы *x1* и *y1* определяют точку P1, а аргументы *x2* и *y2* – точку P2. Дуга, добавляемая в контур, является частью окружности с радиусом *radius*. Начальная точка дуги соответствует точке пересечения с касательной, соединяющей текущую позицию пера и точку P1, а конечная соответствует точке пересечения с касательной, соединяющей точки P1 и P2. Дуга соединяет начальную и конечную точки в кратчайшем направлении. Перед добавлением дуги в контур этот метод добавляет отрезок прямой линии, соединяющий текущую позицию пера с начальной точкой дуги. После вызова этого метода текущей позицией пера является конечная точка дуги, расположенная на линии между точками P1 и P2.

Нарисовать квадрат размером 100×100 с закругленными углами (с разными радиусами), с помощью объекта контекста c можно следующим образом:

```
c.beginPath();
c.moveTo(150, 100);             // Начать с середины верхнего края
c.arcTo(200,100,200,200,40);    // Верхний край и закругленный правый верхний угол
c.arcTo(200,200,100,200,30);    // Правый край и правый нижний угол с закруглением
                                // меньшего радиуса
c.arcTo(100,200,100,100,20);    // Нижний край и закругленный левый нижний угол
c.arcTo(100,100,200,100,10);    // Левый край и закругленный левый верхний угол
c.closePath();                  // Нарисовать отрезок до начальной точки.
c.stroke();                     // Вывести контур
```

void **beginPath()**

Метод `beginPath()` отменяет любое существующее определение контура и начинает новый. После вызова `beginPath()` текущая позиция пера не определена.

При создании в первый раз объекта контекста холста неявно вызывается метод `beginPath()`.

void **bezierCurveTo**(double *cp1x, cp1y,cp2x, cp2y, x, y*)

Метод `bezierCurveTo()` добавляет в текущий подконтур холста кривую Безье третьего порядка. Начальная точка кривой находится в текущей позиции пера, а координаты конечной точки определяются аргументами *x* и *y*. Форма кривой Безье определяется контрольными точками (*cpX1, cpY1*) и (*cpX2, cpY2*). По возвращении из метода текущей позицией становится точка (x, y).

void **clearRect**(double *x, y, width, height*)

Метод `clearRect()` выполняет заливку указанной прямоугольной области черным прозрачным цветом. В отличие от метода `rect()`, он не изменяет текущую позицию пера и текущий контур.

void **clip()**

Вычисляет пересечение внутренней области текущего контура с текущей областью отсечки и использует эту полученную область как новую область отсечки. Обратите внимание, что нет никакого способа увеличить область отсечки. Если область отсечки требуется лишь на время, сначала следует вызвать метод `save()`, чтобы затем с помощью `restore()` восстановить прежнюю область отсечки. По умолчанию область отсечки совпадает с границами холста.

Подобно методу `fill()`, метод `clip()` интерпретирует все контуры как замкнутые и отличает внешнюю и внутреннюю области контура с использованием правила ненулевого числа оборотов.

void **closePath()**

Если текущий подконтур еще не был замкнут, метод closePath() замыкает его добавлением линии, соединяющей текущую и начальную точки контура. После чего начинает новый подконтур (как если бы был вызван метод moveTo()) в текущей точке.

Методы fill() и clip() считают все подконтуры замкнутыми, поэтому явно вызывать метод closePath() необходимо, только если требуется нарисовать замкнутый контур.

ImageData **createImageData(**ImageData *imagedata***)**

Возвращает новый объект ImageData с теми же размерами, что и *data*.

ImageData **createImageData(**double *w*, double *h***)**

Возвращает новый объект ImageData с указанной шириной и высотой. Все пикселы внутри этого нового объекта ImageData инициализируются черным прозрачным цветом (все составляющие цвета и альфа-компонент имеют значение 0).

Аргументы *w* и *h* определяют размеры изображения в CSS-пикселах. Реализациям разрешается отображать один CSS-пиксел в несколько аппаратных пикселов. Свойства width и height возвращаемого объекта ImageData определяют размер изображения в аппаратных пикселах, и их значения могут не совпадать со значениями аргументов *w* и *h*.

CanvasGradient **createLinearGradient(**double *x0*, *y0*, *x1*, *y1***)**

Создает и возвращает новый объект CanvasGradient, который выполняет линейную интерполяцию цветов между заданными начальной и конечной точками. Обратите внимание: этот метод не определяет цвета градиента. Для этих целей следует использовать метод addColorStop() вновь созданного объекта. Чтобы рисовать линии или заполнять фигуры с помощью градиента, необходимо присвоить объект CanvasGradient свойству strokeStyle или fillStyle.

CanvasPattern **createPattern(**Element *image*, string *repetition***)**

Создает и возвращает объект CanvasPattern шаблона, определяющего повторяющееся изображение. Аргумент *image* должен быть элементом <img>, <canvas> или <video>, содержащим изображение, которое будет использоваться как шаблон. Аргумент *repetition* определяет, как будет выкладываться мозаика. Ниже перечислены допустимые значения:

Значение	Смысл
"repeat"	Изображение выкладывается мозаикой в обоих направлениях. Это значение по умолчанию.
"repeat-x"	Изображение выкладывается мозаикой только по оси X.
"repeat-y"	Изображение выкладывается мозаикой только по оси Y.
"no-repeat"	Изображение мозаики не повторяется, а используется однократно.

Чтобы рисовать линии или заполнять фигуры с использованием шаблона, необходимо присвоить объект CanvasPattern свойству strokeStyle или fillStyle.

CanvasGradient **createRadialGradient(**double *x0*, *y0*, *r0*, *x1*, *y1*, *r1***)**

Создает и возвращает новый объект CanvasGradient, который выполняет радиальную интерполяцию цветов между двумя заданными окружностями. Обратите внимание: этот метод не определяет цвета градиента. Для этих целей следует использовать метод addColorStop() вновь созданного объекта. Чтобы рисовать линии или заполнять фигуры с помощью градиента, необходимо присвоить объект `CanvasGradient` свойству `strokeStyle` или `fillStyle`.

Радиальные градиенты отображаются с использованием цвета со смещением 0 для первой окружности, со смещением 1 для второй окружности и интерполированными цветами (красная, зеленая и синяя составляющие, а также альфа-компонент) для рисования промежуточных окружностей.

void **drawImage**(Element *image*, double *dx*, *dy*, [*dw*, *dh*])

Копирует изображение в аргументе *image* (значением которого должен быть элемент <img>, <canvas> или <video>) на холст, помещая верхний левый угол изображения в точку (*dx*, *dy*). Если указаны аргументы *dw* и *dh*, изображение будет масштабировано так, чтобы оно уместилось в область шириной *dw* пикселов и высотой *dh* пикселов.

void **drawImage**(Element *image*, double *sx*, *sy*, *sw*, *sh*, *dx*, *dy*, *dw*, *dh*)

Эта версия метода `drawImage()` копирует прямоугольную область изображения *image* в заданную область холста. Значением аргумента *image* должен быть элемент <img>, <canvas> или <video>. Координаты точки (*sx*,*sy*) определяют верхний левый угол прямоугольной области в исходном изображении, а аргументы *sw* и *sh* – ширину и высоту этой области. Обратите внимание, что значения аргументов измеряются в CSS-пикселах и на них не влияют действующие преобразования системы координат. Остальные аргументы определяют прямоугольную область холста, куда должно быть скопировано изображение: подробности приводятся в описании версии метода `drawImage()` с пятью аргументами выше. Обратите внимание, что аргументы, определяющие прямоугольную область холста, преобразуются в соответствии с текущей матрицей преобразований.

void **fill**()

Метод `fill()` выполняет заливку текущего контура цветом, градиентом или шаблоном, заданным свойством `fillStyle`. Любой незамкнутый подконтур заполняется так, как если бы для него неявно был вызван метод `closePath()`. (Обратите внимание: это не означает, что вызов этого метода сделает подконтур замкнутым.)

Операция заливки текущего контура не очищает его. Можно сразу вслед за методом `fill()` вызвать метод `stroke()` без повторного определения пути.

Когда контур пересекает сам себя или состоит из нескольких накладывающихся друг на друга подконтуров, метод `fill()` пользуется правилом ненулевого числа оборотов для определения, какие точки находятся внутри, а какие вне контура. Это означает, например, что если контур определяет квадрат внутри окружности и подконтур квадрата нарисован в направлении, противоположном направлению рисования окружности, то область внутри квадрата будет считаться лежащей вне контура и не будет заполняться.

void **fillRect**(double *x*, *y*, *width*, *height*)

Метод `fillRect()` выполняет заливку заданного прямоугольника цветом, градиентом или шаблоном, который задается свойством `fillStyle`.

В отличие от метода `rect()`, метод `fillRect()` не влияет на текущую позицию пера и текущий контур.

void **fillText**(string *text*, double *x*, *y*, [double *maxWidth*])

Метод `fillText()` рисует текст *text*, используя текущие значения свойств `font` и `fillStyle`. Аргументы *x* и *y* определяют координаты, где должен выводиться текст, но интерпретация этих аргументов зависит от свойств `textAlign` и `textBaseline`, соответственно.

Если свойство `textAlign` имеет значение «left» или «start» (по умолчанию), в случае использования направления для письма слева направо (также по умолчанию), или «end» в случае использования письма справа налево, текст выводится правее указанной координаты X. Если свойство `textAlign` имеет значение «center», текст центрируется по горизонтали относительно указанной координаты X. В противном случае (если `textAlign` имеет значение «right», «end» для письма слева направо или «start» для письма справа налево) текст выводится левее указанной координаты X.

Если свойство `textBaseline` имеет значение «alphabetic» (по умолчанию), «bottom» или «ideographic», большинство символов будут нарисованы выше указанной координаты Y. Если свойство `textBaseline` имеет значение «center», текст будет центрироваться по вертикали относительно указанной координаты Y. А если свойство `textBaseline` имеет значение «top» или «hanging», большинство символов будут нарисованы ниже указанной координаты Y.

Необязательный аргумент `maxWidth` определяет максимальную ширину текста. Если текст в аргументе *`text`* окажется шире, чем определено аргументом *`maxWidth`*, он будет нарисован более мелким или более узким шрифтом.

`ImageData` **`getImageData`**`(double` *`sx, sy, sw, sh`*`)`

Аргументы этого метода определяют непреобразованные координаты прямоугольной области холста. Метод копирует пикселы из этой области холста в новый объект `ImageData` и возвращает этот объект. Как получить доступ к составляющим цвета и альфа-компонентам отдельных пикселов, описывается в справочной статье `ImageData`.

Компоненты RGB цвета возвращаемых пикселов не учитывают значение альфа-компонента. Если какая-либо часть запрошенной области оказывается за границами холста, соответствующие пикселы в объекте `ImageData` устанавливаются в черный прозрачный цвет (все компоненты цвета равны нулю). Если для представления одного CSS-пиксела реализация использует несколько аппаратных пикселов, значения свойств `width` и `height` возвращаемого объекта `ImageData` будут отличаться от значений аргументов *`sw`* и *`sh`*.

Подобно методу `Canvas.toDataURL()`, этот метод препятствует утечке информации между доменами. Метод `getImageData()` возвращает объект `ImageData`, только если изображение в холсте имеет общее происхождение с документом; в противном случае он возбуждает исключение. Считается, что холст не имеет общего происхождения с документом, если в нем содержалось изображение (созданное непосредственно с помощью метода `drawImage()` или косвенно, с помощью объекта `CanvasPattern`), имеющее иное происхождение, отличное от происхождения содержащего его документа. Кроме того, считается, что элемент `<canvas>` имеет неясное происхождение, если в нем рисовался текст с использованием веб-шрифтов, имеющих иное происхождение.

`boolean` **`isPointInPath`**`(double` *`x, y`*`)`

Метод `isPointInPath()` возвращает `true`, если указанная точка попадает в пределы текущего контура; в противном случае он возвращает `false`. Указанные координаты интерпретируются в системе координат, не преобразованной с применением текущей матрицы преобразования. Аргумент *`x`* должен иметь значение между 0 и `canvas.width`, а аргумент *`y`* – между 0 и `canvas.height`.

Причина, почему `isPointInPath()` использует непреобразованные координаты, состоит в том, что он предназначен для «проверки попадания»: определения попадания указателя мыши во время щелчка (например) в область холста, ограниченную контуром. Чтобы выполнить проверку попадания, координаты указателя мыши сначала должны быть преобразованы из координат окна в координаты холста. Если экранные размеры холста отличаются от размеров, определяемых атрибутами `width` и `height` (например, в случае установки атрибутов `style.width` и `style.height`), координаты указателя мыши также необходимо привести к масштабу, соответствующему масштабу системы координат холста. Ниже демонстрируется функция, которая может использоваться как обработчик `onclick` элемента `<canvas>` и выполнять необходимые преобразования координат указателя мыши в координаты холста:

```
// Обработчик onclick для тега <canvas>. Предполагается, что текущий контур определен.
function hittest(event) {
    var canvas = this;                 // Вызывается в контексте холста
    var c = canvas.getContext("2d"); // Получить контекст рисования для холста

    // Получить размеры и координаты холста
    var bb = canvas.getBoundingClientRect();

    // Преобразовать координаты указателя мыши в координаты холста
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Залить контур, если пользователь щелкнул в его пределах
    if (c.isPointInPath(x,y)) c.fill();
}
```

void **lineTo**(double *x*, double *y*)

Метод `lineTo()` добавляет прямую линию в текущий подконтур. Линия начинается в текущей позиции пера и заканчивается в точке с координатами (*x*,*y*). Когда этот метод возвращает управление, текущая позиция перемещается в точку (*x*,*y*).

TextMetrics **measureText**(string *text*)

Метод `measureText()` вычисляет ширину текста *text*, которую он займет при рисовании с текущим значением свойства `font`, и возвращает объект `TextMetrics`, содержащий результаты вычислений. На момент написания этих строк возвращаемый объект имел только одно свойство, `width`, а высота текста и параметры описывающего прямоугольника не вычислялись.

void **moveTo**(double *x*, double *y*)

Метод `moveTo()` переносит текущую позицию пера в точку (*x*,*y*) и создает новый подконтур с начальной точкой в этой точке. Если перед этим существовал подконтур, состоящий из единственной точки, этот пустой подконтур удаляется из текущего контура.

void **putImageData**(ImageData *imagedata*, double *dx*, *dy*, [*sx*, *sy*, *sw*, *sh*])

Метод `putImageData()` копирует прямоугольную область из объекта `ImageData` в холст. Он выполняет низкоуровневую операцию копирования пикселов, игнорируя значения свойств `globalCompositeOperation` и `globalAlpha`, а также область отсечки, матрицу преобразования и атрибуты, управляющие отображением тени.

Аргументы *dx* и *dy* определяют координаты *назначения* в холсте. Пикселы из объекта в аргументе *imagedata* будут копироваться в холст, начиная с этой точки. Значения этих аргументов не подвергаются преобразованию с применением текущей матрицы преобразования.

Последние четыре аргумента определяют исходную прямоугольную область в объекте `ImageData`. Скопированы будут только пикселы из указанной прямоугольной области. Если эти аргументы отсутствуют, объект `ImageData` будет скопирован целиком. Если эти аргументы определяют прямоугольник, выходящий за границы объекта `ImageData`, прямоугольник будет обрезан по этим границам. В аргументах *sx* и *sy* допускается передавать отрицательные значения.

Одна из ролей объектов `ImageData` – служить «временным хранилищем» для содержимого холста. Сохранение копии холста (с использованием метода `getImageData()`) позволяет временно наносить на холст изображения и затем восстанавливать прежнее состояние холста с помощью `putImageData()`.

`void` **`quadraticCurveTo`**`(double` *`cpx, cpy, x, y`*`)`

Данный метод добавляет кривую Безье второго порядка в текущий подконтур. Начальная точка кривой находится в текущей позиции, а координаты конечной точки определяются аргументами *x* и *y*. Форма кривой Безье, соединяющей эти две точки, определяется контрольной точкой (*cpX, cpY*). По возвращении из метода текущей позицией становится точка (*x,y*). Обратите также внимание на метод `bezierCurveTo()`.

`void` **`rect`**`(double` *`x, y, w, h`*`)`

Добавляет в контур прямоугольник. Прямоугольник представляет собой отдельный подконтур, который никак не связан ни с одним из имеющихся подконтуров. По возвращении из метода текущей позицией становится точка (*x,y*). Вызов этого метода эквивалентен следующей последовательности вызовов:

```
c.moveTo(x,y);
c.lineTo(x+w, y);
c.lineTo(x+w, y+h);
c.lineTo(x, y+h);
c.closePath();
```

`void` **`restore()`**

Метод снимает с вершины стека значения параметров холста и записывает их в свойства объекта `CanvasRenderingContext2D`, восстанавливая область отсечки и матрицу преобразования. Дополнительные сведения см. в справочной статье `save()`.

`void` **`rotate`**`(double` *`angle`*`)`

Данный метод изменяет текущую матрицу преобразования таким образом, что любые фигуры, нарисованные после вызова этого метода, выглядят повернутыми на указанный угол. Этот метод не выполняет вращение самого элемента <canvas>. Обратите внимание: угол задается в радианах. Чтобы преобразовать градусы в радианы, нужно умножить величину угла на константу `Math.PI` и разделить на число 180.

`void` **`save()`**

Метод `save()` помещает копию текущих параметров холста на вершину стека сохраняемых параметров. Это позволяет внести временные изменения в какие-либо параметры и затем восстановить предыдущие значения вызовом метода `restore()`.

В перечень сохраняемых параметров входят все свойства объекта `CanvasRenderingContext2D` (за исключением доступного только для чтения свойства `canvas`), а также матрица преобразования, которая является результатом вызова методов `rotate()`, `scale()` и `translate()`. Кроме того, в стеке сохраняется область отсечки, созданная методом `clip()`. Однако следует заметить, что текущие контур и позиция пера не входят в данный перечень и этим методом не сохраняются.

`void` **`scale`**`(double` *`sx`*`, double` *`sy`*`)`

Метод `scale()` добавляет преобразование масштаба в текущую матрицу преобразования холста. Масштабирование выполняется отдельно по горизонтали и по вертикали. Например, если передать методу значения 2.0 и 0.5, все последующие фигуры будут иметь в два раза большую ширину и в два раза меньшую высоту по сравнению с тем, как они выглядели бы, если бы они были нарисованы до вызова метода `scale()`. Отрицательные значения аргумента *sx* вызывают зеркальное отражение координат относительно оси Y, а отрицательные значения аргумента *sy* вызывают зеркальное отражение координат относительно оси X.

void **setTransform**(double *a*, *b*, *c*, *d*, *e*, *f*)

Этот метод позволяет напрямую установить матрицу преобразования, не выполняя последовательность вызовов методов `translate()`, `scale()` и `rotate()`. После вызова этого метода новое преобразование будет иметь вид:

```
x'   a c e   x  = ax + cy + e
y' = b d f × y  = bx + dy + f
1    0 0 1   1
```

void **stroke**()

Метод `stroke()` выполняет рисование линий, составляющих текущий контур. Контур определяет лишь геометрию линии, которая должна быть воспроизведена, а визуальное ее представление зависит от значений свойств `strokeStyle`, `lineWidth`, `lineCap`, `lineJoin` и `miterLimit`.

Под термином *stroke* (чертить) понимается вычерчивание линий пером или кистью. Это означает «нарисовать контур». В противовес методу `stroke()`, метод `fill()` выполняет заливку внутренней области без рисования ее контура.

void **strokeRect**(double *x*, *y*, *w*, *h*)

Рисует контур (не выполняя заливку внутренней области) прямоугольника с заданными координатами и размерами. Цвет и толщина линий определяются значениями свойств `strokeStyle` и `lineWidth`. Стиль оформления сопряжений в углах прямоугольника определяется значением свойства `lineJoin`.

В отличие от метода `rect()`, метод `strokeRect()` не оказывает влияния на текущий контур или текущую позицию пера.

void **strokeText**(string *text*, double *x*, *y*, [*maxWidth*])

Метод `strokeText()` действует подобно методу `fillText()`, за исключением того, что он не выполняет заливку отдельных символов в соответствии со значением свойства `fillStyle`, а рисует только контуры каждого символа, учитывая значение свойства `strokeStyle`. Для шрифтов большого размера метод `strokeText()` обеспечивает интересный графический эффект, но на практике для рисования текста чаще используется метод `fillText()`.

void **transform**(double *a*, *b*, *c*, *d*, *e*, *f*)

Аргументы этого метода определяют шесть нетривиальных элементов матрицы T аффинного преобразования размером 3×3:

```
a c e
b d f
0 0 1
```

Метод `transform()` умножает текущую матрицу преобразования на матрицу T и принимает результат в качестве текущей матрицы преобразования:

```
CTM' = CTM × T
```

В терминах универсального метода `transform()` можно реализовать операции смещения, масштабирования и вращения. Чтобы выполнить смещение, можно произвести вызов `transform(1,0,0,1,dx,dy)`. Чтобы выполнить масштабирование – `transform(sx, 0, 0, sy, 0, 0)`. Для выполнения вращения по часовой стрелке на угол x:

```
transform(cos(x),sin(x),-sin(x), cos(x), 0, 0)
```

Чтобы выполнить сдвиг параллельно оси X на множитель `k`, можно произвести вызов `transform(1,0,k,1,0,0)`. Сдвига параллельно оси Y можно добиться вызовом `transform(1,k,0,1,0,0)`.

`void` **`translate`**`(double` *`x`*`, double` *`y`*`)`

Метод `translate()` добавляет горизонтальное и вертикальное смещения в матрицу преобразования холста. Значения аргументов *`x`* и *`y`* добавляются к координатам всех точек, которые затем будут добавляться в контур.

ClientRect

прямоугольник, описанный вокруг элемента

Объект `ClientRect` описывает прямоугольник в системе координат объекта `Window` или видимой области. Объект этого типа, определяющий параметры описанного прямоугольника элемента, возвращает метод `getBoundingClientRect()` объекта `Element`. Объекты `ClientRect` являются статическими: они не изменяются при изменении соответствующих им элементов.

Свойства

`readonly float` **`bottom`**

Координата Y нижней границы прямоугольника относительно видимой области.

`readonly float` **`height`**

Высота прямоугольника в пикселах. В IE версии 8 и ниже это свойство не определено; вместо него следует использовать выражение `bottom-top`.

`readonly float` **`left`**

Координата X левой границы прямоугольника относительно видимой области.

`readonly float` **`right`**

Координата X правой границы прямоугольника относительно видимой области.

`readonly float` **`top`**

Координата Y верхней границы прямоугольника относительно видимой области.

`readonly float` **`width`**

Ширина прямоугольника в пикселах. В IE версии 8 и ниже это свойство не определено; вместо него следует использовать выражение `right-left`.

CloseEvent

определяет, был ли закрыт веб-сокет без ошибок **Event**

Когда закрывается соединение `WebSocket`, в объекте `WebSocket` возбуждается невсплывающее и неотменяемое событие «close», и всем зарегистрированным обработчикам этого события передается объект `CloseEvent`.

Свойства

`readonly boolean` **`wasClean`**

Если соединение `WebSocket` было закрыто управляемым способом, как определяется протоколом веб-сокетов, с подтверждением со стороны клиента и сервера, говорят, что закрытие было выполнено *чисто*, и это свойство имеет значение `true`. Если это свойство имеет значение `false`, веб-сокет мог быть закрыт в результате какой-либо сетевой ошибки.

Comment

HTML- или XML-комментарий **Node**

Узел `Comment` представляет комментарий в HTML- или XML-документе. Содержимое комментария (т. е. текст между `<!--` и `-->`) доступно через свойство `data` или через свойство `nodeValue`, унаследованное от интерфейса `Node`. Создать объект `Comment` можно методом `Document.createComment()`.

Свойства

`string` **`data`**

Текст комментария.

`readonly unsigned long` **`length`**

Количество символов в комментарии.

Методы

`void` **`appendData`**`(string` *`data`*`)`

`void` **`deleteData`**`(unsigned long` *`offset`*`, unsigned long` *`count`*`)`

`void` **`insertData`**`(unsigned long` *`offset`*`, string` *`data`*`)`

`void` **`replaceData`**`(unsigned long` *`offset`*`, unsigned long` *`count`*`, string` *`data`*`)`

`string` **`substringData`**`(unsigned long` *`offset`*`, unsigned long` *`count`*`)`

Узлы `Comment` обладают большинством методов узла `Text`, и эти методы действуют так же, как в случае с узлами `Text`. Они перечислены здесь, но их описание приводится в справочной статье `Text`.

Console

вывод отладочной информации

Современные броузеры (и более ранние версии, с установленными расширениями-отладчиками, такими как Firebug) определяют глобальное свойство `console`, ссылающееся на объект `Console`. Методы этого объекта образуют API для выполнения простых отладочных операций, таких как вывод сообщений в окно консоли (консоль можно открыть выбором пункта меню, такого как Developer Tools (Средства разработчика) или Web Inspector (Веб-консоль).

В настоящее время не существует официального стандарта, определяющего API объекта `Console`, но расширение Firebug для Firefox установило стандарт де-факто и производители броузеров стремятся реализовать прикладной интерфейс Firebug, описанный здесь. Поддержка базовой функции `console.log()` реализована практически повсеместно, но реализация других функций может присутствовать не во всех броузерах. Имейте в виду, что в некоторых старых броузерах свойство `console` определено, только если открыто окно консоли, и сценарии, использующие объект `Console`, когда окно консоли не открыто, будут вызывать появление ошибок.

См. также `ConsoleCommandLine`.

Методы

`void` **`assert`**`(any` *`expression`*`, string` *`message`*`)`

Выводит сообщение об ошибке *`message`* в консоли, если выражение *`expression`* имеет значение `false` или любое ложное значение, такое как `null`, `undefined`, 0 или пустая строка.

void **count**([string *title*])

Выводит строку *title* вместе со счетчиком вызовов данного метода с этой же строкой.

void **debug**(any *message*...)

Действует подобно методу `console.log()`, но помечает вывод, как отладочную информацию.

void **dir**(any *object*)

Выводит в консоли информацию об объекте в виде, позволяющем разработчику проверить свойства или элементы и в интерактивном режиме исследовать вложенные объекты и элементы массивов.

void **dirxml**(any *node*)

Выводит в консоль разметку XML или HTML узла документа.

void **error**(any *message*...)

Действует подобно методу `console.log()`, но помечает вывод как ошибку.

void **group**(any *message*...)

Выводит сообщение *message* подобно методу `log()`, но отображает его как заголовок свертываемой группы отладочных сообщений. Все последующие операции вывода в консоль будут помещать сообщения в эту группу, пока не будет вызван соответствующий метод `groupEnd()`.

void **groupCollapsed**(any *message*...)

Создает новую группу сообщений, но в свернутом состоянии, так что по умолчанию последующие отладочные сообщения будут скрыты.

void **groupEnd**()

Закрывает самую последнюю группу отладочных сообщений, созданную вызовом метода `group()` или `groupCollapsed()`.

void **info**(any *message*...)

Действует подобно методу `console.log()`, но помечает вывод как информационное сообщение.

void **log**(string *format*, any *message*...)

Выводит свои аргументы в консоль. В простейшем случае, когда строка *format* не содержит спецификаторов, начинающихся с символа `%`, метод просто преобразует свои аргументы в строки и выводит их, отделяя друг от друга пробелами. Когда методу передается объект, строка, выведенная в консоль, будет доступна для щелчка мышью и позволит просматривать содержимое объекта.

Для вывода более сложных сообщений данный метод поддерживает простейшие спецификаторы формата функции `printf()` из языка C. Аргументы *message* будут интерполироваться в аргумент *format*, на место последовательностей символов «%s», «%d», «%i», «%f» и «%o», после чего в консоль будет выведена отформатированная строка (со следующими за ней аргументами *message*, для которых отсутствуют спецификаторы в аргументе *format*). Аргументы, соответствующие спецификатору «%s», форматируются как строки. Аргументы, соответствующие спецификаторам «%d» и «%i», форматируются как целые числа. Соответствующие спецификатору «%f» форматируются как вещественные числа, а соответствующие спецификатору «%o» – как объекты, доступные для щелчка мышью.

void **profile**([string *title*])

Запускает профилировщик JavaScript и в начале отчета отображает строку *title*.

void **profileEnd()**

Останавливает профилировщик и выводит отчет с результатами профилирования программного кода.

void **time**(string *name*)

Запускает таймер с именем *name*.

void **timeEnd**(string *name*)

Останавливает таймер с именем *name* и выводит имя и время, прошедшее с момента вызова соответствующего метода `time()`.

void **trace()**

Выводит трассировку стека.

void **warn**(any *message*...)

Действует подобно методу `console.log()`, но помечает вывод как предупреждение.

ConsoleCommandLine

глобальные утилиты для работы с окном консоли

Большинство веб-броузеров поддерживают консоль JavaScript (которую вы, возможно, знаете как «Средства разработчика» («Developer Tools») или «Веб-консоль» («Web Inspector»), которая позволяет вводить одиночные строки программного кода на языке JavaScript. В дополнение к обычным глобальным переменным и функциям клиентского JavaScript командная строка консоли обычно поддерживает полезные свойства и функции, описываемые здесь.

См. также `Console`.

Свойства

readonly Element **$0**

Элемент документа, выбранный последним некоторыми средствами отладчика.

readonly Element **$1**

Элемент документа, выбранный перед элементом `$0`.

Методы

void **cd**(Window *frame*)

Когда документ включает вложенные фреймы, функция `cd()` позволяет переключать глобальные объекты и выполнять последующие команды в области видимости фрейма *frame*.

void **clear()**

Очищает окно консоли.

void **dir**(object *o*)

Выводит свойства или элементы объекта или массива *o*. Действует подобно методу `Console.dir()`.

void **dirxml**(Element *elt*)

Выводит разметку XML или HTML элемента *elt*. Действует подобно методу `Console.dirxml()`.

Element **$**(string *id*)

Краткий псевдоним функции `document.getElementById()`.

`NodeList` **`$$`**`(string` *`selector`*`)`

Возвращает объект, подобный массиву, содержащий все элементы, соответствующие CSS-селектору *`selector`*. Это краткий псевдоним функции `document.querySelectorAll()`. В некоторых броузерах возвращает настоящий мссив, а не объект `NodeList`.

`void` **`inspect`**`(any` *`object`*`, [string` *`tabname`*`])`

Отображает объект *`object`*, при этом может переключаться из консоли на другую вкладку отладчика. Во втором аргументе передается необязательная подсказка, определяющая, как должен отображаться объект *`object`*. Поддерживаются значения: «html», «css», «script» и «dom».

`string[]` **`keys`**`(any` *`object`*`)`

Возвращает массив с именами свойств объекта *`object`*.

`void` **`monitorEvents`**`(Element` *`object`*`, [string` *`type`*`])`

Выводит сообщения о событиях типа *`type`*, доставляемых в объект *`object`*. В число поддерживаемых значений аргумента *`type`* входят: «mouse», «key», «text», «load», «form», «drag» и «contextmenu». Если аргумент *`type`* не указан, выводятся сообщения обо всех событиях в объекте *`object`*.

`void` **`profile`**`(string` *`title`*`)`

Запускает профилировщик программного кода. Действует подобно методу `Console.profile()`.

`void` **`profileEnd()`**

Останавливает профилировщик. Действует подобно методу `Console.profileEnd()`.

`void` **`unmonitorEvents`**`(Element` *`object`*`, [string` *`type`*`])`

Останавливает мониторинг событий типа *`type`* в объекте *`object`*.

`any[]` **`values`**`(any` *`object`*`)`

Возвращает массив значений свойств объекта *`object`*.

CSS2Properties

см. CSSStyleDeclaration

CSSRule

правило в таблице стилей CSS

Описание

Объект `CSSRule` является представлением правила в объекте таблице CSS-стилей `CSSStyleSheet`: он дает информацию о стилях, которые должны применяться к определенному набору элементов документа. Свойство `selectorText` – это строковое представление селектора элемента для данного правила, а свойство `style` – ссылка на объект `CSSStyleDeclaration`, который представляет набор атрибутов стилей, применяемых к выбранным элементам.

Иерархия подтипов `CSSRule` для представления различных видов правил, которые могут появляться в таблицах стилей, определяется в спецификации «CSS Object Model». Свойства, описанные здесь, являются универсальными для типа `CSSRule` и его подтипа `CSSStyleRule`. Правила стилей являются наиболее общими и наиболее важными типами правил в таблицах стилей и наиболее часто используемыми в сценариях.

В IE версии 8 и ниже в объектах `CSSRule` поддерживаются только свойства `selectorText` и `style`.

Константы

`unsigned short` **`STYLE_RULE = 1`**

`unsigned short` **`IMPORT_RULE = 3`**

`unsigned short` **`MEDIA_RULE = 4`**

`unsigned short` **`FONT_FACE_RULE = 5`**

`unsigned short` **`PAGE_RULE = 6`**

`unsigned short` **`NAMESPACE_RULE = 10`**

Это допустимые значения свойства `type`, представленного ниже, и они определяют тип правила. Если свойство `type` имеет какое-либо значение, отличное от 1, объект `CSSRule` получит дополнительные свойства, не описываемые здесь.

Свойства

`string` **`cssText`**

Полный текст данного CSS-правила.

`readonly CSSRule` **`parentRule`**

Правило, если таковое имеется, в котором содержится данное правило.

`readonly CSSStyleSheet` **`parentStyleSheet`**

Таблица стилей, внутри которой содержится данное правило.

`string` **`selectorText`**

Когда свойство `type` имеет значение `STYLE_RULE`, это свойство хранит текст селектора, определяющего элементы документа, к которым применяется это правило.

`readonly CSSStyleDeclaration` **`style`**

Когда свойство `type` имеет значение `STYLE_RULE`, это свойство определяет стили, которые должны применяться к элементам, определяемым свойством `selectorText`. Обратите внимание: несмотря на то что свойство `style` доступно только для чтения, свойства объекта `CSSStyleDeclaration`, на которое оно ссылается, доступны для чтения и записи.

`readonly unsigned short` **`type`**

Тип данного правила. Значением этого свойства могут быть только константы, представленные выше.

CSSStyleDeclaration

набор CSS-атрибутов и их значения

Объект `CSSStyleDeclaration` представляет набор CSS-атрибутов стиля и их значения, и позволяет манипулировать этими атрибутами, используя имена свойств, похожие на имена CSS-свойств. Свойство `style` элемента `HTMLElement` является доступным для чтения и записи объектом `CSSStyleDeclaration` и подобно свойству `style` объекта `CSSRule`. Однако метод `Window.getComputedStyle()` возвращает объект `CSSStyleDeclaration`, свойства которого доступны только для чтения.

Объект `CSSStyleDeclaration` обеспечивает доступ к CSS-атрибутам стиля посредством свойств. Имена этих свойств практически однозначно соответствуют именам CSS-атрибутов, незначительно измененными для соответствия синтаксису языка JavaScript. Имена атрибутов, сконструированные из нескольких слов и содержащие дефисы, такие как «font-family», записываются без дефисов, а каждое слово, кроме первого, начинается с заглавного символа: `fontFamily`. Кроме того, имя атрибута «float» совпадает с зарезервированным словом `float`, поэтому оно преобразовано в имя свойства `cssFloat`.

Обратите внимание, что имеется возможность использовать неизмененные имена CSS-атрибутов, если для доступа к свойствам использовать строки и квадратные скобки.

Свойства

Помимо свойств, описанных выше, объект `CSSStyleDeclaration` имеет два дополнительных свойства:

`string` **`cssText`**

Текстовое представление набора атрибутов стиля и их значений. Текст форматируется, как в таблицах стилей CSS, за исключением селектора элемента и фигурных скобок, окружающих атрибуты и значения.

`readonly unsigned long` **`length`**

Количество пар атрибут/значение, содержащихся в данном объекте `CSSStyleDeclaration`. Объект `CSSStyleDeclaration` является также объектом, подобным массиву, элементами которого являются имена объявленных CSS-атрибутов стиля.

CSSStyleSheet

таблица стилей CSS

Этот интерфейс представляет таблицу стилей CSS. Он обладает свойствами и методами, позволяющими деактивировать таблицу стилей, читать, вставлять и удалять объекты правил `CSSRule`. Объекты `CSSStyleSheet`, которые применяются к документу, являются элементами массива `styleSheets[]` объекта `Document` и также доступны через свойство `sheet` элементов `<style>` и `<link>`, определяющих таблицы стилей или ссылающихся на них.

В IE версии 8 и ниже вместо массива `cssRules[]` используется массив `rules[]`, а вместо стандартных методов `insertRule()` и `deleteRule()` – методы `addRule()` и `removeRule()`.

Свойства

`readonly CSSRule[]` **`cssRules`**

Доступный только для чтения объект, подобный массиву, который хранит объекты `CSSRule`, составляющие таблицу стилей. В IE вместо него используется свойство `rules`.

`boolean` **`disabled`**

Значение `true` означает, что таблица стилей неактивна и не будет применяться к документу. Значение `false` – таблица стилей активна и будет применяться к документу.

`readonly string` **`href`**

URL-адрес таблицы стилей, которая связана с документом, или `null`, если таблица стилей встроена в документ.

`readonly string` **`media`**

Список устройств вывода, к которым применяется данная таблица стилей. Значение этого свойства доступно для чтения и записи и может интерпретироваться как единая строка или как объект, подобный массиву, содержащий имена типов устройств вывода и поддерживающий методы `appendMedium()` и `deleteMedium()`. (Формально значением этого свойства является объект `MediaList`, но он не рассматривается в данном справочнике.)

`readonly Node` **`ownerNode`**

Элемент документа, «владеющий» данной таблицей стилей, или `null`, если таковой отсутствует. См. справочные статьи `Link` и `Style`.

`readonly CSSRule` **`ownerRule`**

Объект `CSSRule` правила (из родительской таблицы стилей), которое привело к включению данной таблицы стилей, или `null`, если таблица стилей была подключена каким-то другим способом. (Обратите внимание, что справочная статья `CSSRule` в этом справочнике описывает только правила стилей и не описывает правила `@import`.)

`readonly CSSStyleSheet` **`parentStyleSheet`**

Таблица стилей, которая включает в себя данную таблицу, или `null`, если данная таблица включена непосредственно в документ.

`readonly string` **`title`**

Заголовок таблицы стилей, если указан. Заголовок может определяться атрибутом `title` элемента `<style>` или `<link>`, который ссылается на эту таблицу стилей.

`readonly string` **`type`**

MIME-тип данной таблицы стилей. Таблицы стилей CSS имеют тип «text/css».

Методы

`void` **`deleteRule`**`(unsigned long` *`index`*`)`

Удаляет правило в позиции *`index`* из массива `cssRules`. В IE версии 8 и ниже следует использовать эквивалентный метод `removeRule()`.

`unsigned long` **`insertRule`**`(string` *`rule`*`, unsigned long` *`index`*`)`

Вставляет (или добавляет в конец) новое CSS-правило (строку, определяющую селектор и стили в фигурных скобках) в позицию *`index`* в массив `cssRules` данной таблицы стилей. В IE версии 8 и ниже следует использовать эквивалентный метод `addRule()` и передавать ему две строки, строку селектора и строку со стилями (без фигурных скобок) в первом и втором аргументах, а позицию *`index`* передавать в третьем аргументе.

DataTransfer

передача данных в операциях буксировки мышью

Когда пользователь выполняет операцию буксировки мышью (drag-and-drop), в источнике или в приемнике (или в обоих сразу, если оба они находятся в окне броузера) возбуждается целая последовательность событий. Вместе с этими событиями передается объект события, имеющий свойство `dataTransfer` (см. `Event`), ссылающееся на объект `DataTransfer`. Объект `DataTransfer` занимает центральное место в любой операции буксировки: источник сохраняет в нем передаваемые данные, а приемник извлекает из него переданные данные. Кроме того, объект `DataTransfer` обеспечивает возможность добиться договоренности между источником и приемником о том, должны ли буксируемые данные копироваться, перемещаться или в приемнике должна быть установлена лишь ссылка на них.

Описываемый здесь API был разработан в корпорации Microsoft для использования в IE и реализован, по крайней мере частично, в других броузерах. Спецификация HTML5 стандартизует базовый API, реализованный в IE. Когда эта книга была уже передана в печать, в стандарт HTML5 было добавлено определение новой версии API, в которой предусмотрено свойство `items`, являющееся объектом, подобным массиву, хранящему объекты `DataTransferItem`. Это достаточно привлекательный и разумный

API, но, поскольку он пока еще не реализован в броузерах, он здесь не рассматривается. Вместо этого в данной статье описываются возможности, которые можно использовать (почти) во всех текущих броузерах. Подробное обсуждение этого замысловатого API представлено в разделе 17.7.

Свойства

`string` **`dropEffect`**

Это свойство определяет тип передачи данных, представленных этим объектом. Свойство должно иметь одно из значений: «none», «copy», «move» или «link». Как правило, объект-приемник устанавливает это свойство в обработчике события «dragenter» или «dragover». Значение этого свойства может также зависеть от клавиш модификаторов, удерживаемых пользователем в процессе буксировки, но это во многом зависит от платформы.

`string` **`effectAllowed`**

Это свойство определяет допустимую комбинацию операций перемещения из числа: копирование, перемещение и создание ссылки. Это свойство обычно устанавливается источником в ответ на событие «dragstart». Допустимыми значениями являются: «none» (ни одна), «copy» (копирование), «copyLink» (копирование и создание ссылки), «copyMove» (копирование и перемещение), «link» (создание ссылки), «linkMove» (создание ссылки и перемещение), «move» (перемещение) и «all» (все). (Обратите внимание, что в мнемониках, определяющих две операции, названия операций следуют в алфавитном порядке.)

`readonly File[]` **`files`**

Если перемещаемые данные являются одним или более файлами, это свойство будет ссылаться на массив или на объект, подобный массиву, содержащий объекты `File`.

`readonly string[]` **`types`**

Это объект, подобный массиву, содержащий строки, которые определяют MIME-типы данных, сохраняемых в объекте `DataTransfer` (устанавливается методом `setData()`, если источник располагается в броузере, или каким-либо другим механизмом, если источник находится за пределами броузера). Объект, подобный массиву, хранящий типы, должен иметь метод `contains()` для проверки присутствия определенной строки. Однако некоторые броузеры передают в этом свойстве истинный массив, и в этом случае для проверки можно использовать метод `indexOf()`.

Методы

`void` **`addElement(`**`Element` *`element`*`)`

Сообщает броузеру элемент *`element`*, который можно использовать для воспроизведения визуального эффекта, который будет наблюдать пользователь во время буксировки. Обычно этот метод вызывается буксируемым источником, но он может быть реализован или иметь хоть какой-нибудь эффект не во всех броузерах.

`void` **`clearData(`**`[string` *`format`*`])`

Удаляет все данные в формате *`format`*, которые были добавлены методом `setData()`.

`string` **`getData(`**`string` *`format`*`)`

Возвращает отбуксированные данные в формате *`format`*. Если в аргументе *`format`* передано значение «text» (без учета регистра символов), возвращаются данные в формате «text/plain». А если передано значение «url» (без учета регистра символов), возвращаются данные в формате «text/uri-list». Этот метод вызывается приемником в ответ на событие «drop» в конце операции буксировки.

void **setData**(string *format*, string *data*)

Принимает данные *data* для передачи и MIME-тип этих данных в аргументе *format*. Источник вызывает этот метод в ответ на событие «dragstart» в начале операции буксировки. Он не может вызываться из какого-либо другого обработчика событий. Если источник способен представить данные более чем в одном формате, он может вызвать этот метод несколько раз, чтобы определить значения для каждого поддерживаемого формата.

void **setDragImage**(Element *image*, long *x*, long *y*)

Определяет изображение *image* (обычно элемент <img>), которое должно отображаться как визуальное представление буксируемых данных. Аргументы *x* и *y* определяют смещение указателя мыши относительно изображения. Этот метод может вызываться только буксируемым источником, в ответ на событие «dragstart».

DataView

реализует чтение/запись значений в ArrayBuffer **ArrayBufferView**

DataView – это подтип ArrayBufferView, который служит оберткой для ArrayBuffer (или фрагмента ArrayBuffer) и определяет методы чтения/записи 1-, 2- и 4-байтовых целых со знаком и без знака, а также 4- и 8-байтовых вещественных чисел в буфере. Методы поддерживают прямой (big-endian) и обратный (little-endian) порядок следования байтов. См. также TypedArray.

Конструктор

new **DataView**(ArrayBuffer *buffer*, [unsigned long *byteOffset*], [unsigned long *byteLength*])

Этот конструктор создает новый объект DataView, обеспечивающий доступ на чтение и запись к байтам в буфере или во фрагменте буфера. При вызове с одним аргументом создает представление всего буфера. При вызове с двумя аргументами создает представление, простирающееся от байта с номером *byteOffset* до конца буфера. И при вызове с тремя аргументами создает представление длиной *byteLength*, начинающееся с байта с номером *byteOffset*.

Методы

Следующие методы читают или записывают числовые значения в буфер ArrayBuffer, представленный объектом DataView. Имя метода определяет тип читаемого или записываемого числового значения. Все методы, выполняющие чтение или запись более одного байта, принимают необязательный последний аргумент *littleEndian*. Если этот аргумент отсутствует или имеет значение false, используется прямой (big-endian) порядок следования байтов, когда старшие байты читаются или записываются первыми. Однако если этот аргумент имеет значение true, используется обратный (little-endian) порядок следования байтов.

float **getFloat32**(unsigned long *byteOffset*, [boolean *littleEndian*])

Интерпретирует 4 байта, начиная с позиции *byteOffset*, как вещественное число и возвращает его.

double **getFloat64**(unsigned long *byteOffset*, [boolean *littleEndian*])

Интерпретирует 8 байтов, начиная с позиции *byteOffset*, как вещественное число и возвращает его.

short **getInt16**(unsigned long *byteOffset*, [boolean *littleEndian*])

Интерпретирует 2 байта, начиная с позиции *byteOffset*, как целое число со знаком и возвращает его.

long **getInt32**(unsigned long *byteOffset*, [boolean *littleEndian*])

Интерпретирует 4 байта, начиная с позиции *byteOffset*, как целое число со знаком и возвращает его.

byte **getInt8**(unsigned long *byteOffset*)

Интерпретирует байт в позиции *byteOffset*, как целое число со знаком и возвращает его.

unsigned short **getUint16**(unsigned long *byteOffset*,[boolean *littleEndian*])

Интерпретирует 2 байта, начиная с позиции *byteOffset*, как целое число без знака и возвращает его.

unsigned long **getUint32**(unsigned long *byteOffset*, [boolean littleEndian])

Интерпретирует 4 байта, начиная с позиции *byteOffset*, как целое число без знака и возвращает его.

unsigned byte **getUint8**(unsigned long *byteOffset*)

Интерпретирует байт в позиции *byteOffset*, как целое число без знака и возвращает его.

void **setFloat32**(unsigned long *byteOffset*, float *value*, [boolean *littleEndian*])

Преобразует значение *value* в 4-байтовое вещественное представление и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setFloat64**(unsigned long *byteOffset*, double *value*, [boolean *littleEndian*])

Преобразует значение *value* в 8-байтовое вещественное представление и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setInt16**(unsigned long *byteOffset*, short *value*, [boolean *littleEndian*])

Преобразует значение *value* в 2-байтовое целочисленное представление и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setInt32**(unsigned long *byteOffset*, long *value*, [boolean *littleEndian*])

Преобразует значение *value* в 4-байтовое целочисленное представление и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setInt8**(unsigned long *byteOffset*, byte *value*)

Преобразует значение *value* в 1-байтовое целочисленное представление и записывает полученный байт в буфер, в позицию *byteOffset*.

void **setUint16**(unsigned long *byteOffset*, unsigned short *value*,[boolean *littleEndian*])

Преобразует значение *value* в 2-байтовое целочисленное представление без знака и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setUint32**(unsigned long *byteOffset*, unsigned long *value*, [boolean *littleEndian*])

Преобразует значение *value* в 4-байтовое целочисленное представление без знака и записывает полученные байты в буфер, начиная с позиции *byteOffset*.

void **setUint8**(unsigned long *byteOffset*, octet *value*)

Преобразует значение *value* в 1-байтовое целочисленное представление без знака и записывает полученный байт в буфер в позицию *byteOffset*.

Document

HTML- или XML-документ **Node**

Объект Document – это узел Node, играющий роль корня дерева документа. Свойство documentElement объекта представляет корневой элемент Element документа. Узел Document может иметь несколько дочерних узлов (таких как узлы Comment и DocumentType), но он имеет только один дочерний узел типа `Element`, хранящий все содержимое документа.

Проще всего доступ к объекту `Document` получить через свойство `document` объекта `Window`. Доступ к объекту `Document` можно также получить через свойство `contentDocument` элемента `IFrame` или через свойство `ownerDocument` любого узла типа `Node`.

Большинство свойств объекта `Document` обеспечивают доступ к элементам документа или к другим важным объектам, связанным с документом. Некоторые методы объекта `Document` обеспечивают похожие возможности, предоставляя средства поиска элементов в дереве документа. Многие другие методы являются «фабричными методами», используемыми для создания элементов и других объектов.

Объекты `Document`, подобно объектам `Element`, которые они содержат, являются целевыми объектами событий. Они реализуют методы, определяемые интерфейсом `EventTarget`, а также поддерживают множество свойств обработчиков событий.

Создать новый объект `Document` можно с помощью методов `createDocment()` и `createHTMLDocument()` объекта `DOMImplementation`:

```
document.implementation.createHTMLDocument("New Doc");
```

Кроме того, можно загрузить HTML- или XML-файл из сети и преобразовать его в объект `Document`. См. описание свойства `responseXML` объекта `XMLHttpRequest`.

Справочная статья `HTMLDocument`, имевшаяся в предыдущем издании этой книги, теперь объединена с этой справочной статьей. Обратите внимание, что некоторые свойства, методы и обработчики, описываемые здесь, могут использоваться только при работе с HTML-документами и не могут применяться к XML-документам.

Свойства

Помимо свойств, перечисленных ниже, в качестве свойств документа можно также использовать значения атрибутов `name` элементов `<iframe>`, `<form>` и `<img>`. Значениями этих свойств являются соответствующие объекты `Element` или `NodeList` этих элементов. Однако для элементов `<iframe>` свойство ссылается на объект `Window` элемента `<iframe>`. Подробности приводятся в разделе 15.2.2.

`readonly Element` **`activeElement`**

Элемент документа, владеющий в настоящий момент фокусом ввода.

`Element` **`body`**

В HTML-документах это свойство ссылается на элемент `<body>`. (Однако в документах, определяющих элементы `<frameset>`, это свойство ссылается на самый внешний элемент `<frameset>`.)

`readonly string` **`characterSet`**

Кодировка символов документа.

`string` **`charset`**

Кодировка символов документа. Это свойство подобно свойству `characterSet`, но его значение можно изменить, чтобы сменить кодировку документа.

`readonly string` **`compatMode`**

Это свойство содержит строку «BackCompat», если документ отображается в «режиме совместимости» для обратной совместимости с очень старыми броузерами. Иначе это свойство содержит строку «CSS1Compat».

`string` **`cookie`**

Это свойство позволяет читать, создавать, изменять и удалять cookies, которые применяются к данному документу. *Cookies* – это небольшие блоки именованных данных, хранимых веб-броузером. Они позволяют броузеру «запомнить» некоторые данные, которые могут быть введены в одной странице, а использоваться в другой, или повторно использовать предпочтения пользователя между вызовами страниц в рамках сеанса. Данные в cookies автоматически передаются между веб-броузером и веб-сервером, благодаря чему серверные сценарии могут читать и изменять значения в cookies. Клиентские сценарии на языке JavaScript также могут читать и изменять значения в cookies, используя это свойство. Обратите внимание, что это свойство доступно для чтения/записи, но в целом прочитанное из него значение доступно только для чтения, как и записанное в него значение. Подробности приводятся в разделе 20.2.

`readonly string` **`defaultCharset`**

Кодировка символов в броузере по умолчанию.

`readonly Window` **`defaultView`**

Объект `Window` броузера, в котором отображается данный документ.

`string` **`designMode`**

Если это свойство имеет значение «on», документ целиком доступен для редактирования. Если это свойство имеет значение «off», документ целиком недоступен для редактирования. (Но при этом доступными для редактирования могут быть отдельные элементы с установленным свойством `contenteditable`.) Подробности приводятся в разделе 15.10.4.

`string` **`dir`**

В HTML-документах это свойство соответствует атрибуту `dir` элемента `<html>`. То есть это то же самое значение, что и `documentElement.dir`.

`readonly DocumentType` **`doctype`**

Узел `DocumentType`, представляющий объявление `<!DOCTYPE>` документа.

`readonly Element` **`documentElement`**

Корневой элемент документа. В HTML-документах это свойство всегда является объектом `Element`, представляющим тег `<html>`. Этот корневой элемент также доступен через массив `childNodes[]`, унаследованный от `Node`, но он необязательно будет первым элементом этого массива. См. также описание свойства `body`.

`string` **`domain`**

Доменное имя сервера, откуда был загружен документ, или `null`, если он отсутствует. В отдельных случаях это свойство может использоваться для ослабления политики общего происхождения и получения доступа к документам из родственных доменов. Подробности приводятся в разделе 13.6.2.1.

`readonly HTMLCollection` **`embeds`**

Объект, подобный массиву, содержащий элементы `<embed>`, присутствующие в документе.

`readonly HTMLCollection` **`forms`**

Объект, подобный массиву, содержащий все элементы `Form`, присутствующие в документе.

`readonly Element` **`head`**

В HTML-документах это свойство ссылается на элемент `<head>`.

`readonly HTMLCollection` **`images`**

Объект, подобный массиву, содержащий все элементы `Image`, присутствующие в документе.

`readonly DOMImplementation` **`implementation`**

Объект `DOMImplementation` для данного документа.

`readonly string` **`lastModified`**

Содержит дату и время последнего изменения документа. Это значение поставляется в HTTP-заголовке «Last-Modified», который может передаваться веб-сервером по требованию.

`readonly HTMLCollection` **`links`**

Объект, подобный массиву, содержащий все гиперссылки, присутствующие в документе. Этот объект `HTMLCollection` содержит все элементы `<a>` и `<area>`, имеющие атрибут `href`, но не включает элементы `<link>`. См. также `Link`.

`readonly Location` **`location`**

Синоним свойства `Window.location`.

`readonly HTMLCollection` **`plugins`**

Синоним свойства `embeds`.

`readonly string` **`readyState`**

Это свойство содержит строку «loading», пока продолжается загрузка документа, и строку «complete» по ее окончании. Когда это свойство получает значение «complete», броузер возбуждает событие «readystatechange» в объекте `Document`.

`readonly string` **`referrer`**

URL-адрес документа, ссылающегося на данный документ, или `null`, если документ был открыт не через гиперссылку, или если веб-сервер не сообщил адрес документа, содержащего ссылку на данный документ. Это свойство позволяет получить доступ к HTTP-заголовку `referer`. Обратите внимание на правописание: имя заголовка включает три символа «r», а имя JavaScript-свойства – четыре.

`readonly HTMLCollection` **`scripts`**

Объект, подобный массиву, содержащий все элементы `<script>`, присутствующие в документе.

`readonly CSSStyleSheet[]` **`styleSheets`**

Коллекция объектов, представляющих все таблицы стилей, встроенные в документ, или на которые есть ссылки из него. В HTML-документах они включают таблицы стилей, определенные с помощью тегов `<link>` и `<style>`.

`string` **`title`**

Текстовое содержимое тега `<title>` данного документа.

`readonly string` **`URL`**

URL-адрес документа. Это значение часто совпадает со значением свойства `location.href`. Однако если сценарий изменит идентификатор фрагмента документа (свойство `location.hash`), свойства `location` и `URL` будут ссылаться на разные URL-адреса. Не путайте свойство `Document.URL` со свойством `Window.URL`.

Методы

`Node` **`adoptNode`**`(Node` *`node`*`)`

Удаляет узел *node* из любого документа, частью которого он являлся на момент вызова, и записывает в свойство `ownerDocument` узла ссылку на текущий документ, подготавливая его к добавлению в текущий документ. Похожий на него метод `importNode()` копирует узел из другого документа, не удаляя его.

void **close()**

Закрывает поток вывода документа, открытый методом open(), заставляя вывести все буферизованные данные.

Comment **createComment(**string *data***)**

Создает и возвращает новый узел Comment с указанным содержимым.

DocumentFragment **createDocumentFragment()**

Создает и возвращает новый пустой узел DocumentFragment.

Element **createElement(**string *localName***)**

Создает и возвращает новый пустой узел Element с указанным именем тега. В HTML-документах символы в имени тега преобразуются в верхний регистр.

Element **createElementNS(**string *namespace*, string *qualifiedName***)**

Создает и возвращает новый пустой узел Element. Первый аргумент определяет идентификатор URI пространства имен элемента, а второй – префикс пространства имен, двоеточие и имя тега элемента.

Event **createEvent(**string *eventInterface***)**

Создает и возвращает неинициализированный объект Event искусственного события. Аргумент определяет тип события и должен быть строкой, такой как «Event», «UIEvent», «MouseEvent», «MessageEvent» и так далее. После создания объекта Event можно инициализировать его свойства, доступные только для чтения, вызовом соответствующих методов инициализации, таких как initEvent(), initUIEvent(), initMouseEvent() и так далее. Большая часть методов инициализации не рассматриваются в этой книге, но описание простейшего из них приводится в справочной статье Event.initEvent(). После создания и инициализации объекта искусственного события его можно отправить вызовом метода dispatchEvent() интерфейса EventTarget. Искусственные события всегда имеют значение false в свойстве isTrusted.

ProcessingInstruction **createProcessingInstruction(**string *target*, string *data***)**

Создает и возвращает новый узел ProcessingInstruction со строками *target* и *data*.

Text **createTextNode(**string *data***)**

Создает и возвращает новый узел Text, представляющий текст *text*.

Element **elementFromPoint(**float *x*, float *y***)**

Возвращает самый глубоко вложенный элемент Element с оконными координатами (*x*, *y*).

boolean **execCommand(**string *commandId*, [boolean *showUI*, [string *value*]]**)**

Выполняет команду редактирования с именем в аргументе *commandId* в любом доступном для редактирования элементе, в котором находится текстовый курсор. Спецификация HTML5 определяет следующие команды:

bold
createLink
delete
formatBlock
forwardDelete
insertImage
insertHTML
insertLineBreak
insertOrderedList
insertUnorderedList
insertParagraph
insertText
italic
redo
selectAll
subscript
superscript
undo
unlink
unselect

Некоторые из этих команд (такие как createLink) требуют аргумент *value*. Если второй аргумент метода execCommand() имеет значение false, то значение аргумента команды определяется третьим аргументом метода. Иначе броузер предложит пользователю ввести необходимое значение. Подробнее о методе `execCommand()` рассказывается в разделе 15.10.4.

`Element` **`getElementById`**`(string` *`elementId`*`)`

Отыскивает в документе узел `Element` с атрибутом `id`, значение которого совпадает со значением аргумента *`elementId`*, и возвращает этот элемент. Если такой элемент не найден, он возвращает `null`. Значение атрибута `id` предполагается уникальным в пределах документа, а если этот метод найдет более одного элемента с указанным значением *`elementId`*, то он вернет первый из них. Этот метод важен и часто используется, т. к. обеспечивает простой способ получения объекта `Element`, представляющего определенный элемент в документе. Обратите внимание: имя этого метода оканчивается суффиксом «Id», а не «ID».

`NodeList` **`getElementsByClassName`**`(string` *`classNames`*`)`

Возвращает объект, подобный массиву, содержащий элементы, в атрибуте `class` которых присутствуют все имена классов, указанные в *`classNames`*. Аргумент *`classNames`* может содержать единственное имя класса или список имен классов, разделенных пробелами. Возвращаемый объект `NodeList` – «живой» объект, который автоматически обновляется при изменении документа. Элементы в возвращаемом объекте `NodeList` располагаются в том же порядке, в каком они присутствуют в документе. Обратите внимание, что этот метод также определен в объекте `Element`.

`NodeList` **`getElementsByName`**`(string` *`elementName`*`)`

Возвращает «живой», доступный только для чтения объект, подобный массиву, содержащий элементы со значением *`elementName`* в атрибуте `name`. Если искомые элементы отсутствуют, возвращается объект `NodeList` со значением 0 в свойстве `length`.

`NodeList` **`getElementsByTagName`**`(string` *`qualifiedName`*`)`

Возвращает доступный только для чтения объект, подобный массиву, содержащий все узлы `Element` из документа, имеющие указанное имя тега, в том порядке, в котором они располагаются в исходном тексте документа. Объект `NodeList` – «живой», т. е. его содержимое по необходимости автоматически обновляется, если элементы с указанным именем тега добавляются или удаляются из документа. Сравнение с именами тегов HTML-элементов выполняется без учета регистра символов. Как особый случай, имени тега «*» соответствуют все элементы документа.

Обратите внимание: интерфейс `Element` определяет метод с тем же именем, который выполняет поиск по поддереву документа.

`NodeList` **`getElementsByTagNameNS`**`(string` *`namespace`*`, string` *`localName`*`)`

Этот метод работает точно так же, как `getElementsByTagName()`, но при его использовании имя тега указывается как комбинация идентификатора URI пространства имен и локального имени тега в этом пространстве имен.

`boolean` **`hasFocus`**`()`

Возвращает `true`, если окно с данным документом владеет фокусом ввода (или, если это окно не является окном верхнего уровня, все его родители владеют фокусом ввода).

`Node` **`importNode`**`(Node` *`node`*`, boolean` *`deep`*`)`

Получает узел, определенный в другом документе, и возвращает копию узла, подходящую для вставки в данный документ. Если аргумент *`deep`* имеет значение `true`, копируются также все потомки узла. Исходный узел и его потомки никак не модифицируются. В полученной копии свойство `ownerDocument` устанавливается равным данному документу, а `parentNode` – `null`, поскольку копия пока не вставлена в документ. Обработчики событий, зарегистрированные в исходном узле или дереве, не копируются. См. также `adoptNode()`.

`Window` **`open`**`(string` *`url`*`, string` *`name`*`, string` *`features`*`, [boolean` *`replace`*`])`

Когда метод `open()` документа вызывается с тремя и более аргументами, он действует подобно методу `open()` объекта `Window`. См. также `Window`.

`Document` **`open`**`([string` *`type`*`], [string` *`replace`*`])`

При вызове с двумя и менее аргументами этот метод стирает текущий документ и начинает новый (используя существующий объект `Document`, который является возвращаемым значением). После вызова `open()` можно использовать методы `write()` и `writeln()`, чтобы вывести содержимое в поток документа, и метод `close()`, чтобы завершить создание документа и заставить броузер отобразить его. Подробности приводятся в разделе 15.10.2.

Новый документ будет являться HTML-документом, если аргумент *type* отсутствует или имеет значение «text/html». Иначе будет создан простой текстовый документ. Если аргумент *replace* имеет значение `true`, новый документ заменит прежний в истории посещений броузера.

Этот метод не должен вызываться сценарием или обработчиком событий, являющимся частью переписываемого документа, т.к. сам сценарий или обработчик также будет переписан.

`boolean` **`queryCommandEnabled`**`(string` *`commandId`*`)`

Возвращает `true`, если в настоящий момент можно передать команду *commandId* методу `execCommand()`, и `false` – в противном случае. Например, бессмысленно передавать команду «undo», когда нечего отменять. Подробности приводятся в разделе 15.10.4.

`boolean` **`queryCommandIndeterm`**`(string` *`commandId`*`)`

Возвращает `true`, если *commandId* находится в состоянии, для которого `queryCommandState()` не может вернуть какое-то определенное значение. Команды, определяемые спецификацией HTML5, не могут находиться в неопределенном состоянии, но команды, определяемые броузером, – могут. Подробности приводятся в разделе 15.10.4.

`boolean` **`queryCommandState`**`(string` *`commandId`*`)`

Возвращает состояние команды *commandId*. Некоторые команды редактирования, такие как «bold» и «italic», имеют состояние `true`, если под текстовым курсором или в выделенной области находится текст, набранный курсивом, и `false` – в противном случае. Однако большинство команд не имеют состояния, и для них этот метод всегда возвращает `false`. Подробности приводятся в разделе 15.10.4.

`boolean` **`queryCommandSupported`**`(string` *`commandId`*`)`

Возвращает `true`, если броузер поддерживает указанную команду, и `false` – в противном случае. Подробности приводятся в разделе 15.10.4.

`string` **`queryCommandValue`**`(string` *`commandId`*`)`

Возвращает состояние указанной команды в виде строки. Подробности приводятся в разделе 15.10.4.

`Element` **`querySelector`**`(string` *`selectors`*`)`

Возвращает первый элемент в данном документе, соответствующий CSS-селекторам *selectors* (это может быть единственный CSS-селектор или группа селекторов, разделенных запятыми).

`NodeList` **`querySelectorAll`**`(string` *`selectors`*`)`

Возвращает объект, подобный массиву, содержащий все элементы `Element` в данном документе, соответствующие селекторам *selectors* (это может быть единственный CSS-селектор или группа селекторов, разделенных запятыми). В отличие от объектов `NodeList`, возвращаемых методом `getElementsByTagName()` и аналогичными ему, объект `NodeList`, возвращаемый этим методом, является статическим и содержит элементы, соответствующие селекторам, существовавшие на момент вызова метода.

`void` **`write`**`(string` *`text...`*`)`

Добавляет текст *`text`* в конец документа. Этот метод можно использовать в ходе загрузки документа для вставки текста в позицию тега `<script>` или после вызова метода `open()`. Подробности приводятся в разделе 15.10.2.

`void` **`writeln`**`(string` *`text...`*`)`

Этот метод похож на `HTMLDocument.write()` за исключением того, что за добавляемым текстом следует символ перевода строки, что может быть удобным, например, при формировании содержимого тега `<pre>`.

События

Непосредственно в объекте `Document` броузеры возбуждают немного событий, но события, генерируемые в элементах, будут всплывать до объекта `Document`, вмещающего их. По этой причине объект `Document` поддерживает все свойства обработчиков событий, перечисленные в справочной статье `Element`. Подобно элементам `Element`, объект `Document` реализует методы интерфейса `EventTarget`.

Непосредственно в объекте `Document` броузеры возбуждают два события. Когда изменяется значение свойства `readyState`, броузеры возбуждают событие «readystatechange». Зарегистрировать обработчик этого события можно с помощью свойства `onreadystatechange`. Кроме того, броузеры возбуждают событие «DOMContentLoaded» (раздел 17.4), когда дерево документа будет готово к манипуляциям (но до окончания загрузки внешних ресурсов). Однако для регистрации обработчиков этого события необходимо использовать метод интерфейса `EventTarget`, потому что в объекте `Document` отсутствует свойство с именем `onDOMContentLoaded`.

DocumentFragment

смежные узлы и их поддеревья **Node**

Интерфейс `DocumentFragment` представляет часть (фрагмент) документа. Если говорить конкретно, то он представляет список смежных узлов документа и всех их потомков, но без общего родительского узла. Узлы `DocumentFragment` никогда не являются частью дерева документа, а унаследованное свойство `parentNode` в них всегда равно `null`. Однако особенность узлов `DocumentFragment` делает их очень полезными: когда поступает запрос на вставку `DocumentFragment` в дерево документа, вставляется не сам узел `DocumentFragment`, а все его дочерние узлы. Поэтому интерфейс `DocumentFragment` хорош в качестве временного хранилища для узлов, которые требуется вставить в документ все сразу.

Создать новый пустой узел `DocumentFragment` можно с помощью метода `Document.createDocumentFragment()`.

Поиск элементов в узле `DocumentFragment` можно выполнить с помощью методов `querySelector()` и `querySelectorAll()`, которые действуют так же, как одноименные методы объекта `Document`.

Методы

`Element` **`querySelector`**`(string` *`selectors`*`)`

См. `Document.querySelector()`.

`NodeList` **`querySelectorAll`**`(string` *`selectors`*`)`

См. `Document.querySelectorAll()`.

DocumentType

объявление <!DOCTYPE> документа **Node**

Этот редко используемый интерфейс представляет объявление `<!DOCTYPE>` документа. Свойство `doctype` объекта `Document` хранит узел `DocumentType` этого документа. Узлы `DocumentType` являются неизменяемыми, и нет никакого способа изменить их.

Узлы `DocumentType` используются для создания новых объектов `Document` с помощью метода `DOMImplementation.createDocument()`. Новый объект `DocumentType` можно создать с помощью `DOMImplementation.createDocumentType()`.

Свойства

`readonly string` **`name`**

Имя типа документа. Это идентификатор, который следует непосредственно за объявлением `<!DOCTYPE>` в начале документа и совпадает с именем тега корневого элемента документа. В HTML-документах это свойство содержит значение «html».

`readonly string` **`publicId`**

Внешний идентификатор DTD или пустая строка, если идентификатор не указан.

`readonly string` **`systemId`**

Системный идентификатор DTD или пустая строка, если идентификатор не указан.

DOMException

исключение, возбужденное Web API

Большинство прикладных интерфейсов в клиентском JavaScript возбуждают исключение `DOMException`, когда им требуется сообщить об ошибке. Более подробная информация об ошибке содержится в свойствах `code` и `name` объекта. Обратите внимание, что исключение `DOMException` может быть возбуждено при чтении или изменении свойства или при вызове метода объекта.

`DOMException` не является подклассом типа `Error` базового JavaScript, но функционально похож на него, и некоторые броузеры добавляют в него свойство `message` для совместимости с классом `Error`.

Константы

```
unsigned short INDEX_SIZE_ERR = 1
unsigned short HIERARCHY_REQUEST_ERR = 3
unsigned short WRONG_DOCUMENT_ERR = 4
unsigned short INVALID_CHARACTER_ERR = 5
unsigned short NO_MODIFICATION_ALLOWED_ERR = 7
unsigned short NOT_FOUND_ERR = 8
unsigned short NOT_SUPPORTED_ERR = 9
unsigned short INVALID_STATE_ERR = 11
unsigned short SYNTAX_ERR = 12
unsigned short INVALID_MODIFICATION_ERR = 13
unsigned short NAMESPACE_ERR = 14
unsigned short INVALID_ACCESS_ERR = 15
unsigned short TYPE_MISMATCH_ERR = 17
```

unsigned short **SECURITY_ERR = 18**
unsigned short **NETWORK_ERR = 19**
unsigned short **ABORT_ERR = 20**
unsigned short **URL_MISMATCH_ERR = 21**
unsigned short **QUOTA_EXCEEDED_ERR = 22**
unsigned short **TIMEOUT_ERR = 23**
unsigned short **DATA_CLONE_ERR = 25**

Эти константы определяют допустимые значения для свойства `code`. Имена констант достаточно полно объясняют причины, повлекшие исключение

Свойства

unsigned short **code**

Значение одной из констант, перечисленных выше, определяющих тип исключения.

string **name**

Имя типа исключения. Это будет имя одной из констант, перечисленных выше, в виде строки.

DOMImplementation

глобальные методы DOM

Объект `DOMImplementation` определяет методы, не относящиеся к какому-либо конкретному объекту `Document`, а являющиеся «глобальными» для реализации DOM. Ссылку на объект `DOMImplementation` можно получить через свойство `implementation` любого объекта `Document`.

Методы

Document **createDocument**(string *namespace*, string *qualifiedName*, DocumentType *doctype*)

Создает и возвращает новый объект `Document` XML-документа. Если указан аргумент *qualifiedName*, создается корневой элемент с этим именем и добавляется в документ как значение его свойства `documentElement`. Если *qualifiedName* включает префикс пространства имен и двоеточие, *пространство имен* должно быть представлено идентификатором URI, уникально идентифицирующим его. Если аргумент *doctype* содержит значение, отличное от `null`, свойству `ownerDocument` этого объекта `DocumentType` присваивается вновь созданный документ, а узел `DocumentType` добавляется в новый документ.

DocumentType **createDocumentType**(string *qualifiedName*, *publicId*, *systemId*)

Создает новый узел `DocumentType`, представляющий объявление `<!DOCTYPE>`, который можно передать методу `createDocument()`.

Document **createHTMLDocument**(string *title*)

Создает новый объект `HTMLDocument` с готовым деревом документа, включающий указанный заголовок. Значением свойства `documentElement` возвращаемого объекта является элемент `<html>`, и этот корневой элемент содержит вложенные теги `<head>` и `<body>`. Элемент `<head>` в свою очередь включает вложенный элемент `<title>` с дочерним текстовым узлом, содержащим строку *title*.

DOMSettableTokenList

список лексем с настраиваемым строковым значением **DOMTokenList**

Объект `DOMSettableTokenList` является подтипом `DOMTokenList`, имеющим дополнительное свойство `value`, которому можно присвоить сразу полный список лексем.

Свойство `classList` объекта `Element` является ссылкой на объект `DOMTokenList`, который представляет множество лексем в свойстве `className` в виде строки. Если потребуется присвоить свойству `classList` сразу все лексемы, можно просто присвоить новую строку свойству `className`. Свойство `sandbox` элемента `IFrame` несколько отличается. Это свойство и HTML-атрибут, на который оно опирается, было определено в спецификации HTML5, и потому не было никакой необходимости использовать смесь из старого строкового представления и объекта `DOMTokenList`. Это свойство просто определено как объект `DOMSettableTokenList`, что позволяет читать его и присваивать ему значение, как если бы это была простая строка, или использовать методы и интерпретировать его как множество лексем. Свойство `htmlFor` объекта `Output` и свойство `audio` объекта `Video` также являются объектами `DOMSettableTokenList`.

Свойства

`string` **`value`**

Представление множества лексем в виде строки, в которой лексемы разделены пробелами. Это свойство позволяет обрабатывать множество как единственную строку. Однако обычно не возникает необходимости использовать это свойство явно: при использовании объекта `DOMSettableTokenList` в контексте, где требуется строка, возвращается именно это строковое значение. А если выполнить присваивание строки объекту `DOMSettableTokenList`, строка неявно будет записана в это свойство.

DOMTokenList

множество лексем, разделенных пробелами

Объект `DOMTokenList` представляет результат разбора строки со списком лексем, разделенных пробелами как, например, свойство `className` объекта `Element`. Объект `DOMTokenList`, как следует из его имени, является списком, точнее объектом, подобным массиву, со свойством `length`, который можно индексировать для получения доступа к отдельным лексемам. Но более важно, что он определяет методы `contains()`, `add()`, `remove()` и `toggle()`, позволяющие работать с ним как со множеством лексем. Если использовать объект `DOMTokenList` в строковом контексте, он будет интерпретироваться как строка со списком лексем, разделенных пробелами.

Свойство `classList` объектов `Element`, определяемое спецификацией HTML5, является объектом `DOMTokenList` в броузерах, поддерживающих это свойство. И это единственный объект `DOMTokenList`, который вам наверняка придется часто использовать на практике. См. также `DOMSettableTokenList`.

Свойства

`readonly unsigned long` **`length`**

`DOMTokenList` – это объект, подобный массиву; данное свойство определяет количество уникальных лексем, содержащихся в нем.

Методы

`void` **`add`**`(string` *`token`*`)`

Если `DOMTokenList` еще не содержит лексему *`token`*, она будет добавлена в конец списка.

`boolean` **`contains`**`(string` *`token`*`)`

Возвращает `true`, если объект `DOMTokenList` содержит лексему *`token`*, или `false` – в противном случае.

`string` **`item`**`(unsigned long` *`index`*`)`

Возвращает лексему по указанному индексу или `null`, если индекс *`index`* выходит за границы массива. Объект `DOMTokenList` можно также индексировать непосредственно, не прибегая к этому методу.

`void` **`remove`**`(string` *`token`*`)`

Если `DOMTokenList` содержит лексему *`token`*, этот метод удалит ее. Иначе он ничего делать не будет.

`boolean` **`toggle`**`(string` *`token`*`)`

Если `DOMTokenList` содержит лексему *`token`*, этот метод удалит ее. Иначе – добавит.

Element

элемент документа **Node, EventTarget**

Объект `Element` представляет элементы HTML- или XML-документа. Свойство `tagName` определяет имя тега или тип элемента. Стандартные HTML-атрибуты элемента доступны в виде JavaScript-свойств объекта `Element`. Доступ к атрибутам, включая XML-атрибуты и нестандартные HTML-атрибуты, можно также получить с помощью методов `getAttribute()` и `setAttribute()`. Содержимое элемента `Element` доступно через свойства, унаследованные от объекта `Node`. Если требуется выполнить операции с элементами, связанными отношениями с данным элементом, можно воспользоваться свойствами `children`, `firstElementChild`, `nextElementSibling` и другими похожими свойствами.

Существует несколько способов получения объектов `Element` из документов. Свойство `documentElement` объекта `Document` содержит ссылку на корневой элемент этого документа, такой как элемент `<html>` HTML-документа. В HTML-документах имеются похожие на него свойства `head` и `body` – они ссылаются на элементы `<head>` и `<body>` документа. Чтобы отыскать элемент документа по уникальному значению атрибута `id`, можно воспользоваться методом `Document.getElementById()`. Как описывается в разделе 15.2, объекты `Element` можно также получить с помощью методов `getElementsByTagName()`, `getElementsByClassName()` и `querySelectorAll()`. Наконец, с помощью метода `Document.createElement()` можно создавать новые объекты `Element` для вставки в документ.

Веб-броузеры возбуждают в элементах документа множество различных видов событий, и объекты `Element` определяют множество свойств обработчиков событий. Кроме того, объекты `Element` определяют методы интерфейса `EventTarget` (подробнее о нем в справочной статье `EventTarget`), позволяющие добавлять и удалять обработчики событий.

Справочная статья `HTMLElement`, имевшаяся в предыдущем издании этой книги, теперь объединена с этой справоч ной статьей. Обратите внимание, что некоторые свойства, методы и обработчики, описываемые здесь, могут использоваться только при работе с HTML-документами и не могут применяться к XML-документам.

Свойства

Помимо свойств, перечисленных ниже, HTML-атрибуты HTML-элементов доступны в виде свойств объектов `Element`. HTML-теги и их стандартные атрибуты перечислены в конце этой справочной статьи.

`readonly Attr[]` **`attributes`**

Объект, подобный массиву, содержащий объекты `Attr`, представляющие HTML- или XML-атрибуты данного элемента. Однако в общем случае объекты `Element` обеспечивают доступ к атрибутам, как к JavaScript-свойствам, поэтому почти никогда не возникает необходимости использовать массив `attributes[]`.

`readonly unsigned long` **`childElementCount`**

Количество дочерних элементов (не дочерних узлов), которые имеет данный элемент.

`readonly HTMLCollection` **`children`**

Объект, подобный массиву, содержащий дочерние элементы (исключая дочерние узлы, не являющиеся элементами `Element`, такие как `Text` и `Comment`).

`readonly DOMTokenList` **`classList`**

Значение атрибута `class` элемента представляет собой список имен классов, разделенных пробелами. Данное свойство позволяет получить доступ к отдельным элементам этого списка и определяет методы чтения, добавления, удаления и переключения отдельных имен классов. Подробности смотрите в справочной статье `DOMTokenList`.

`string` **`className`**

Это свойство представляет атрибут `class` элемента. В языке JavaScript `class` является зарезервированным словом, поэтому свойство получило имя `className`, а не `class`. Обратите внимание, что такое имя этого свойства[1] может вводить в заблуждение, поскольку атрибут `class` часто включает более одного имени класса.

`readonly long` **`clientHeight`**
`readonly long` **`clientWidth`**

Если данный элемент является корневым элементом (`document.documentElement`), эти свойства возвращают внутренние размеры окна, т. е. размеры видимой области, куда не входят полосы прокрутки и другие элементы управления, добавляемые броузером. Иначе эти свойства возвращают размеры содержимого элемента плюс отступы.

`readonly long` **`clientLeft`**
`readonly long` **`clientTop`**

Эти свойства возвращают расстояние в пикселах между левой или верхней рамкой элемента и левой или верхней границей отступов. Обычно это просто толщина левой или верхней стороны рамки, но сюда также может входить ширина полос прокрутки, если они отображаются вдоль левого или верхнего края элемента.

`CSSStyleDeclaration` **`currentStyle`**

Это свойство, реализованное только в IE, является представлением каскадного набора всех CSS-свойств, применяемых к элементу. В IE версии 8 и ниже его можно использовать как замену стандартному методу `Window.getComputedStyle()`.

`readonly object` **`dataset`**

С любым HTML-элементом можно связать любые значения, присваивая их атрибутам, имена которых начинаются с префикса «data-». Данное свойство `dataset` представляет множество атрибутов с данными и упрощает работу с ними.

Значение этого свойства ведет себя как обычный объект. Каждое свойство объекта соответствует одному атрибуту с данными. Если элемент имеет атрибут с именем `data-x`, объект `dataset` получит свойство с именем `x`, и `dataset.x` будет возвращать то же значение, что и вызов `getAttribute("data-x")`.

Операции чтения и присваивания значений свойствам объекта `dataset` будут читать и присваивать значения соответствующим атрибутам с данными этого элемента. Оператор `delete` можно использовать для удаления атрибутов с данными, а цикл `for/in` – для их перечисления.

---

[1] Имя свойства указывается в форме единственного числа. – *Прим. перев.*

readonly Element **firstElementChild**

Это свойство подобно свойству `firstChild` объекта `Node`, но оно игнорирует узлы `Text` и `Comment` и возвращает только элементы типа `Element`.

string **id**

Значение атрибута `id`. Никакие два элемента в одном документе не должны иметь одинаковые значения атрибута `id`.

string **innerHTML**

Доступная для чтения и записи строка, которая определяет текст разметки HTML или XML, содержащийся внутри элемента, за исключением открывающего и закрывающего тегов самого элемента. Операция чтения этого свойства возвращает содержимое элемента в виде строки HTML- или XML-текста. Операция записи замещает содержимое элемента представлением HTML- или XML-текста после его синтаксического разбора.

readonly boolean **isContentEditable**

Это свойство имеет значение `true`, если элемент доступен для редактирования, и `false` – в противном случае. Элемент может быть доступен для редактирования вследствие установки свойства `contenteditable` в нем или в его родителе, или вследствие установки свойства `designMode` вмещающего объекта `Document`.

string **lang**

Значение атрибута `lang`, определяющее код языка для содержимого элемента.

readonly Element **lastElementChild**

Это свойство подобно свойству `lastChild` объекта `Node`, но оно игнорирует узлы `Text` и `Comment` и возвращает только элементы типа `Element`.

readonly string **localName**

Локальное имя данного элемента без префикса. Значение этого свойства отличается от значения атрибута `tagName`, которое может включать префикс пространства имен, если таковой имеется (и все символы которого для HTML-элементов преобразуются в верхний регистр).

readonly string **namespaceURI**

URL-адрес, формально определяющий пространство имен данного элемента. Может иметь значение `null` или содержать строку, такую как «http://www.w3.org/1999/xhtml».

readonly Element **nextElementSibling**

Это свойство подобно свойству `nextSibling` объекта `Node`, но оно игнорирует узлы `Text` и `Comment` и возвращает только элементы типа `Element`.

readonly long **offsetHeight**
readonly long **offsetWidth**

Высота и ширина элемента и всего его содержимого в пикселах, включая отступы и рамки, но без учета полей.

readonly long **offsetLeft**
readonly long **offsetTop**

Координаты X и Y верхнего левого угла CSS-рамки элемента относительно контейнерного элемента `offsetParent`.

readonly Element **offsetParent**

Ссылается на контейнерный элемент, определяющий систему координат, относительно которой измеряются свойства `offsetLeft` и `offsetTop`. Для большинства элементов свойство `offsetParent` ссылается на вмещающий их объект `<body>`. Однако если контейнерный элемент имеет динамическое позиционирование, ссылка на него становится значением свойства `offsetParent` динамически позиционируемого элемента, а если элемент располагается в таблице, значением свойства `offsetParent` может быть ссылка на элемент `<td>`, `<th>` или `<table>`. Подробности приводятся в разделе 15.8.5.

`string` **`outerHTML`**

Разметка HTML или XML, определяющая данный элемент и его содержимое. Если присвоить этому свойству строку, она заменит данный элемент (и все его содержимое) результатом синтаксического разбора нового значения как фрагмента HTML- или XML-документа.

`readonly string` **`prefix`**

Префикс пространства имен для данного элемента. Обычно это свойство содержит значение `null`. Исключение составляют XML-документы, в которых используются пространства имен.

`readonly Element` **`previousElementSibling`**

Это свойство подобно свойству `previousSibling` объекта `Node`, но оно игнорирует узлы `Text` и `Comment` и возвращает только элементы типа `Element`.

`readonly long` **`scrollHeight`**

`readonly long` **`scrollWidth`**

Общая высота и ширина элемента в пикселах. Когда элемент имеет полосы прокрутки (например, потому что был установлен CSS-атрибут `overflow`), значения этих свойств отличаются от значений свойств `offsetHeight` и `offsetWidth`, которые просто содержат размеры видимой части элемента.

`long` **`scrollLeft`**

`long` **`scrollTop`**

Число пикселов, на которое элемент был прокручен за левую или верхнюю границу. Обычно эти свойства полезны только для элементов с полосами прокрутки, у которых, например, CSS-атрибут `overflow` имеет значение `auto`. В элементе `<html>` (см. `Document.documentElement`) эти свойства определяют общую величину прокрутки всего документа. Обратите внимание: эти свойства не определяют величину прокрутки в теге `<iframe>`. Этим свойствам можно присваивать значения, чтобы выполнять прокрутку элемента или всего документа. Подробности приводятся в разделе 15.8.5.

`readonly CSSStyleDeclaration` **`style`**

Значение атрибута `style`, задающее встроенные CSS-стили для элемента. Обратите внимание: значение этого свойства является не строкой, а объектом, свойства которого соответствуют CSS-атрибутам и доступны для чтения и записи. Подробности см. в справочной статье об объекте `CSSStyleDeclaration`.

`readonly string` **`tagName`**

Имя тега элемента. Для элементов HTML-документа имя тега возвращается в верхнем регистре независимо от регистра символов в исходном тексте документа, т. е. элемент `<p>` будет иметь в свойстве `tagName` строку «P». XML-документы чувствительны к регистру, и имя тега возвращается в точности в том виде, в каком оно записано в исходном тексте документа. Это свойство имеет то же значение, что и свойство `nodeName` интерфейса `Node`.

`string` **`title`**

Значение атрибута `title` элемента. Многие броузеры отображают значение этого атрибута в виде всплывающей подсказки при наведении указателя мыши на элемент.

Методы

`void` **`blur()`**

Передает фокус ввода элементу `body` вмещающего объекта `Document`.

`void` **`click()`**

Имитирует щелчок мышью на элементе. Если в случае щелчка на данном элементе должно что-то происходить (например, переход по ссылке), вызов этого метода также приведет к выполнению этих действий. В противном случае этот метод просто сгенерирует событие «click» в элементе.

`void` **`focus()`**

Передает фокус ввода в данный элемент.

`string` **`getAttribute`**`(string` *`qualifiedName`*`)`

Метод `getAttribute()` возвращает значение указанного атрибута для элемента или `null`, если такого атрибута не существует. Обратите внимание, что объекты, представляющие HTML-элементы, определяют JavaScript-свойства, соответствующие стандартным HTML-атрибутам, поэтому надобность в этом методе возникает только при необходимости обратиться к нестандартным атрибутам. В HTML-документах сравнение имен атрибутов выполняется без учета регистра символов.

В XML-документах значения атрибутов недоступны непосредственно как свойства элемента, и к ним надо обращаться путем вызова этого метода. Для XML-документов, в которых используются пространства имен, когда в имя атрибута включается префикс пространства имен и двоеточие, может потребоваться использовать метод `getAttributeNS()` или `getAttributeNodeNS()`.

`string` **`getAttributeNS`**`(string` *`namespace`*`, string` *`localName`*`)`

Этот метод действует так же, как метод `getAttribute()`, кроме того, что атрибут задается комбинацией URI пространства имен и локального имени, определенного в данном пространстве имен.

`ClientRect` **`getBoundingClientRect()`**

Возвращает объект `ClientRect`, описывающий прямоугольник, ограничивающий данный элемент.

`ClientRect[]` **`getClientRects()`**

Возвращает объект, подобный массиву, содержащий объекты `ClientRects`, которые описывают один или более прямоугольников, ограничивающих данный элемент. (Чтобы точно описать область окна, занимаемую строчными элементами, размещающимися в нескольких строках, обычно требуется более одного прямоугольника.)

`NodeList` **`getElementsByClassName`**`(string` *`classNames`*`)`

Возвращает объект, подобный массиву, содержащий вложенные элементы, в которых значение атрибута `class` включает все имена классов `classNames`. Аргумент `classNames` может содержать имя одного класса или список нескольких имен классов, разделенных пробелами. Возвращаемый объект `NodeList` является «живым» и автоматически обновляется при изменении документа. Элементы в возвращаемом объекте `NodeList` располагаются в том же порядке, в каком они присутствуют в документе. Обратите внимание, что этот метод также определен в объекте `Document`.

`NodeList` **`getElementsByTagName`**`(string` *`qualifiedName`*`)`

Выполняет обход всех вложенных элементов и возвращает «живой» объект `NodeList` узлов `Element`, представляющих все элементы документа с указанным именем тега. Элементы в возвращаемом объекте `NodeList` располагаются в том же порядке, в каком они присутствуют в исходном документе. Обратите внимание, что объект `Document` также имеет метод `getElementsByTagName()`, действующий подобным образом, но выполняющий обход всего документа, а не только элементов, вложенных в данный элемент.

`NodeList` **`getElementsByTagNameNS`**`(string` *`namespace`*`, string` *`localName`*`)`

Этот метод действует подобно методу `getElementsByTagName()`, за исключением того, что имя тега требуемых элементов указывается как комбинация URI пространства имен и локального имени в этом пространстве имен.

`boolean` **`hasAttribute`**`(string` *`qualifiedName`*`)`

Возвращает `true`, если этот элемент имеет атрибут с указанным именем, или `false` в противном случае. В HTML-документах имена атрибутов нечувствительны к регистру символов.

`boolean` **`hasAttributeNS`**`(string` *`namespace`*`, string` *`localName`*`)`

Этот метод действует так же, как метод `hasAttribute()`, за исключением того, что атрибут задается комбинацией URI пространства имен и локального имени в этом пространстве имен.

`void` **`insertAdjacentHTML`**`(string` *`position`*`, string` *`text`*`)`

Вставляет разметку HTML *`text`* в позицию *`position`* относительно данного элемента. Аргумент *`position`* может иметь одно из следующих строковых значений:

Значение аргумента position	Описание
beforebegin	Вставляет текст перед открывающим тегом элемента
afterend	Вставляет текст после закрывающего тега элемента
afterbegin	Вставляет текст сразу после открывающего тега элемента
beforeend	Вставляет текст непосредственно перед закрывающим тегом элемента

`Element` **`querySelector`**`(string` *`selectors`*`)`

Возвращает первый вложенный элемент, соответствующий CSS-селекторам *`selectors`* (это может быть единственный CSS-селектор или группа селекторов, разделенных запятыми).

`NodeList` **`querySelectorAll`**`(string` *`selectors`*`)`

Возвращает объект, подобный массиву, содержащий все элементы, вложенные в данный элемент, которые соответствуют селекторам *`selectors`* (это может быть единственный CSS-селектор или группа селекторов, разделенных запятыми). В отличие от объекта `NodeList`, возвращаемого методом `getElementsByTagName()`, объект `NodeList`, возвращаемый этим методом, является статическим: он содержит набор элементов, соответствовавших селекторам на момент вызова метода.

`void` **`removeAttribute`**`(string` *`qualifiedName`*`)`

Удаляет из элемента атрибут с указанным именем. Попытки удалить несуществующие атрибуты просто игнорируются. В HTML-документах имена атрибутов нечувствительны к регистру символов.

`void` **`removeAttributeNS`**`(string` *`namespace`*`, string` *`localName`*`)`

Метод `removeAttributeNS()` действует так же, как метод `removeAttribute()`, за исключением того, что удаляемый атрибут задается URI пространства имен и локального имени.

void **scrollIntoView**([boolean *top*])

Если HTML-элемент в настоящий момент находится за пределами окна, этот метод прокрутит документ так, что элемент окажется в пределах окна. Аргумент *top* является необязательным и подсказывает методу, должен ли элемент оказаться ближе к верхнему или к нижнему краю окна. Если он равен true или отсутствует, броузер старается выполнить прокрутку так, чтобы элемент оказался ближе к верхнему краю окна. Если он равен false, броузер старается выполнить прокрутку так, чтобы элемент оказался ближе к нижнему краю окна. Для элементов, принимающих фокус ввода, таких как элементы Input, метод focus() неявно выполняет точно такую же операцию прокрутки. См. также описание метода scrollTo() объекта Window.

void **setAttribute**(string *qualifiedName*, string *value*)

Присваивает указанное значение атрибуту с указанным именем. Если атрибут с таким именем еще не существует, в элемент добавляется новый атрибут. В HTML-документах перед присваиванием значения символы в имени атрибута преобразуются в нижний регистр. Обратите внимание: в HTML-документе JavaScript-свойства, соответствующие всем стандартным HTML-атрибутам, определяются объектами HTMLElement. Поэтому данный метод обычно используется лишь для доступа к нестандартным атрибутам.

void **setAttributeNS**(string *namespace*, string *qualifiedName*, string *value*)

Этот метод действует так же, как метод setAttribute(), за исключением того, что имя атрибута указывается как комбинация URI пространства имен и квалифицированного имени, состоящего из префикса пространства имен, двоеточия и локального имени в этом пространстве имен.

Обработчики событий

Объекты Element, представляющие HTML-элементы, определяют достаточно много свойств обработчиков событий. Достаточно присвоить функцию любому свойству из числа перечисленных ниже, и эта функция будет вызываться при возникновении события данного типа в элементе (или по достижении элемента в результате всплытия). Для регистрации обработчиков событий можно также использовать методы, определяемые интерфейсом EventTarget.

Большинство событий всплывают вверх по дереву документа до узла Document и затем передаются объекту Window. Поэтому все свойства обработчиков событий, перечисленные ниже, определены также в объектах Document и Window. Однако объект Window обладает достаточно большим количеством собственных обработчиков событий, и свойства, помеченные звездочкой в таблице ниже, в объекте Window имеют другой смысл. По историческим причинам обработчики событий, зарегистрированные посредством HTML-атрибутов в элементе <body>, регистрируются в объекте Window, а это означает, что свойства, помеченные звездочкой, в элементе <body> имеют другой смысл. См. также Window.

Многие события, перечисленные здесь, возбуждаются только в HTML-элементах определенных типов. Но, так как многие из этих событий всплывают вверх по дереву документа, свойства обработчиков событий определены во всех элементах. Мультимедийные события, введенные спецификацией HTML5, которые возбуждаются в тегах <audio> и <video>, не всплывают, поэтому они описываются в справочной статье MediaElement. Аналогично некоторые события форм, введенные спецификацией HTML5, также не всплывают и описываются в справочной статье FormControl.

Обработчик события	Вызывается, когда...
`onabort`	загрузка ресурса прервана по требованию пользователя
`onblur*`	элемент потерял фокус ввода
`onchange`	пользователь изменил содержимое или состояние элемента формы (возбуждается по окончании редактирования, а не по каждому нажатию клавиши)
`onclick`	элемент активирован щелчком мыши или каким-то другим способом
`oncontextmenu`	контекстное меню готово к отображению, обычно в результате щелчка правой кнопки
`ondblclick`	выполнен двойной щелчок мышью
`ondrag`	буксировка продолжается (возбуждается в элементе-источнике)
`ondragend`	буксировка завершена (возбуждается в элементе-источнике)
`ondragenter`	буксируемые данные оказались над элементом (возбуждается в элементе-приемнике)
`ondragleave`	буксируемые данные вышли за границы элемента (возбуждается в элементе-приемнике)
`ondragover`	буксировка продолжается (возбуждается в элементе-приемнике)
`ondragstart`	пользователь начал операцию буксировки (возбуждается в элементе-источнике)
`ondrop`	пользователь завершил буксировку (возбуждается в элементе-приемнике)
`onerror*`	возникла ошибка загрузки ресурса (обычно вследствие сетевой ошибки)
`onfocus*`	элемент получил фокус ввода
`oninput`	выполнен ввод в элемент формы (возбуждается значительно чаще, чем `onchange`)
`onkeydown`	пользователь нажал клавишу
`onkeypress`	в результате нажатия на клавишу сгенерирован печатаемый символ
`onkeyup`	пользователь отпустил клавишу
`onload*`	загрузка ресурса (например, изображения для элемента `<img>`) завершена
`onmousedown`	пользователь нажал кнопку мыши
`onmousemove`	пользователь переместил указатель мыши
`onmouseout`	указатель мыши вышел за границы элемента
`onmouseover`	указатель мыши оказался над элементом
`onmouseup`	пользователь отпустил кнопку мыши
`onmousewheel`	пользователь повернул колесико мыши
`onreset`	выполнен сброс формы
`onscroll*`	выполнена прокрутка элемента, имеющего полосы прокрутки
`onselect`	пользователь выделил текст в элементе формы
`onsubmit`	выполнена отправка формы

HTML-элементы и атрибуты

Этот раздел включает справочные статьи для следующих типов HTML-элементов:

Элемент	Справочная статья	Элемент	Справочная статья
`<audio>`	`Audio`	`<output>`	`Output`
`<button>`, `<input type="button">`	`Button`	`<progress>`	`Progress`
`<canvas>`	`Canvas`	`<script>`	`Script`
`<fieldset>`	`FieldSet`	`<select>`	`Select`
`<form>`	`Form`	`<style>`	`Style`
`<iframe>`	`Iframe`	`<td>`	`TableCell`
`<img>`	`Image`	`<tr>`	`TableRow`
`<input>`	`Input`	`<tbody>`, `<tfoot>`, `<thead>`	`TableSection`
`<label>`	`Label`	`<table>`	`Table`
`<a>`, `<area>`, `<link>`	`Link`	`<textarea>`	`TextArea`
`<meter>`	`Meter`	`<video>`	`Video`
`<option>`	`Option`		

HTML-элементы, для которых отсутствуют собственные справочные статьи, относятся к числу тех, чьи свойства просто соответствуют HTML-атрибутам элементов. Ниже перечислены атрибуты, допустимые в любых HTML-элементах и потому являющиеся свойствами всех объектов `Element`:

Атрибут	Описание
`accessKey`	Горячая комбинация клавиш на клавиатуре
`class`	CSS-класс: см. описание свойств `className` и `classList` выше.
`contentEditable`	Доступно ли содержимое элемента для редактирования.
`contextMenu`	Значение атрибута `id` элемента `<menu>`, который должен отображаться как контекстное меню. На момент написания этих строк поддерживался только в IE.
`dir`	Направление письма: «ltr» (слева направо) или «rtl» (справа налево).
`draggable`	Логический атрибут, устанавливаемый в элементах, которые могут служить элементами-источниками для API буксировки.
`dropzone`	Логический атрибут, устанавливаемый в элементах, которые могут служить элементами-приемниками для API буксировки.
`hidden`	Логический атрибут, устанавливаемый в элементах, которые не должны отображаться.
`id`	Уникальный идентификатор элемента.
`lang`	Основной язык текста в элементе.
`spellcheck`	Требуется ли проверять орфографию в текстовом содержимом элемента.
`style`	Встроенные стили CSS элемента. См. описание свойства `style` выше.
`tabIndex`	Определяет порядковый номер элемента в операциях передачи фокуса ввода.
`title`	Текст всплывающей подсказки для элемента.

Следующие HTML-элементы не имеют атрибутов, кроме тех, что перечислены выше:

- `<abbr>`
- `<address>`
- `<article>`
- `<aside>`
- `<b>`
- `<bdi>`
- `<bdo>`
- `<br>`
- `<caption>`
- `<cite>`
- `<code>`
- `<datalist>`
- `<dd>`
- `<dfn>`
- `<div>`
- `<dl>`
- `<dt>`
- `<em>`
- `<figcaption>`
- `<figure>`
- `<footer>`
- `<h1>`
- `<h2>`
- `<h3>`
- `<h4>`
- `<h5>`
- `<h6>`
- `<head>`
- `<header>`
- `<hgroup>`
- `<hr>`
- `<i>`
- `<kbd>`
- `<legend>`
- `<mark>`
- `<nav>`
- `<noscript>`
- `<p>`
- `<pre>`
- `<rp>`
- `<rt>`
- `<ruby>`
- `<s>`
- `<samp>`
- `<section>`
- `<small>`
- `<span>`
- `<strong>`
- `<sub>`
- `<summary>`
- `<sup>`
- `<tbody>`
- `<tfoot>`
- `<thead>`
- `<title>`
- `<tr>`
- `<ul>`
- `<var>`
- `<wbr>`

Остальные HTML-элементы и поддерживаемые ими атрибуты перечислены ниже. Обратите внимание, что в этой таблице перечислены только атрибуты, отличающиеся от глобальных атрибутов, перечисленных выше. Также отметьте, что эта таблица включает элементы, для которых имеются отдельные справочные статьи:

Элемент	Атрибуты
`<a>`	href, target, ping, rel, media, hreflang, type
`<area>`	alt, coords, shape, href, target, ping, rel, media, hreflang, type
`<audio>`	src, preload, autoplay, loop, controls
`<base>`	href, target
`<blockquote>`	cite
`<body>`	onafterprint, onbeforeprint, onbeforeunload, onblur, onerror, onfocus, onhashchange, onload, onmessage, onoffline, ononline, onpagehide, onpageshow, onpopstate, onredo, onresize, onscroll, onstorage, onundo, onunload
`<button>`	autofocus, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, name, type, value
`<canvas>`	width, height
`<col>`	span
`<colgroup>`	span
`<command>`	type, label, icon, disabled, checked, radiogroup
`<del>`	cite, datetime
`<details>`	open
`<embed>`	src, type, width, height
`<fieldset>`	disabled, form, name
`<form>`	accept-charset, action, autocomplete, enctype, method, name, novalidate, target
`<html>`	manifest
`<iframe>`	src, srcdoc, name, sandbox, seamless, width, height
`<img>`	alt, src, usemap, ismap, width, height
`<input>`	accept, alt, autocomplete, autofocus, checked, dirname, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, height, list, max, maxlength, min, multiple, name, pattern, placeholder, readonly, required, size, src, step, type, value, width
`<ins>`	cite, datetime

Элемент	Атрибуты
`<keygen>`	autofocus, challenge, disabled, form, keytype, name
`<label>`	form, for
`<li>`	value
`<link>`	href, rel, media, hreflang, type, sizes
`<map>`	name
`<menu>`	type, label
`<meta>`	name, http-equiv, content, charset
`<meter>`	value, min, max, low, high, optimum, form
`<object>`	data, type, name, usemap, form, width, height
`<ol>`	reversed, start
`<optgroup>`	disabled, label
`<option>`	disabled, label, selected, value
`<output>`	for, form, name
`<param>`	name, value
`<progress>`	value, max, form
`<q>`	cite
`<script>`	src, async, defer, type, charset
`<select>`	autofocus, disabled, form, multiple, name, required, size
`<source>`	src, type, media
`<style>`	media, type, scoped
`<table>`	summary
`<td>`	colspan, rowspan, headers
`<textarea>`	autofocus, cols, disabled, form, maxlength, name, placeholder, readonly, required, rows, wrap
`<th>`	colspan, rowspan, headers, scope
`<time>`	datetime, pubdate
`<track>`	default, kind, label, src, srclang
`<video>`	src, poster, preload, autoplay, loop, controls, width, height

ErrorEvent

необработанное исключение в фоновом потоке выполнения **Event**

Когда в фоновом потоке выполнения, который представляет объект `Worker`, возникает необработанное исключение, которое также не было обработано функцией `onerror` в объекте `WorkerGlobalScope`, оно вызывает невсплывающее событие «error» в объекте `Worker`. Вместе с событием обработчику передается объект события `ErrorEvent`, содержащий информацию об исключении. Вызов метода `preventDefault()` объекта `ErrorEvent` (или возврат значения `false` из обработчика события) предотвратит дальнейшее распространение события по объемлющим потокам выполнения и также может предотвратить вывод сообщения об ошибке в консоли.

Свойства

`readonly string` **`filename`**

URL-адрес файла со сценарием на языке JavaScript, в котором возникло исключение.

`readonly unsigned long` **`lineno`**

Номер строки в этом файле, в которой возникло исключение.

`readonly string` **`message`**

Сообщение с описанием исключения.

Event

информация о стандартных событиях, события IE и jQuery

Когда вызывается обработчик события, ему передается объект `Event`, чьи свойства содержат дополнительную информацию о событии, такую как тип события и элемент, в котором оно возникло. С помощью методов этого объекта `Event` можно управлять распространением события. Все современные броузеры реализуют стандартную модель событий, кроме IE, который в версии 8 и ниже определяет свою собственную, не совместимую с другими броузерами модель. Эта справочная статья описывает стандартные свойства и методы объекта события, альтернативные им свойства и методы, поддерживаемые в IE, а также охватывает объект события, реализованный в библиотеке jQuery, который имитирует стандартный объект события в IE. Более подробно о событиях рассказывается в главе 17, а о поддержке событий в библиотеке jQuery – в разделе 19.4.

В стандартной модели событий для различных типов событий определены различные типы объектов событий, связанных с ними: по событиям мыши, например, обработчикам передается объект `MouseEvent` со свойствами, имеющими отношение к мыши, а по событиям клавиатуры передается объект `KeyEvent` со свойствами, имеющими отношение к клавиатуре. Оба типа, `MouseEvent` и `KeyEvent`, наследуют общий суперкласс `Event`. Однако в моделях событий IE и jQuery по всем событиям, возникающим в объектах `Element`, передаются объекты единого типа `Event`. Свойства объекта `Event`, имеющие отношение к клавиатуре, не будут иметь сколько-нибудь осмысленные значения в обработчике события мыши, но сами свойства будут определены в объекте. Для простоты в этой справочной статье не рассматривается иерархия событий и описываются свойства всех событий, которые могут получить объекты `Element` (и затем всплыть до объектов `Document` и `Window`).

Первоначально практически все события в клиентских сценариях генерировались в элементах документа, поэтому такое смешивание свойств, имеющих отношение к документу, в одном объекте выглядит вполне естественным. Однако HTML5 и связанные с ним стандарты определяют множество новых типов событий, которые генерируются в объектах, не являющихся элементами документа. Для этих типов событий часто определяются собственные объекты событий, и этим объектам посвящены собственные справочные статьи. См. статьи `BeforeUnloadEvent`, `CloseEvent`, `ErrorEvent`, `HashChangeEvent`, `MessageEvent`, `PageTransitionEvent`, `PopStateEvent`, `ProgressEvent` и `StorageEvent`.

Большинство из этих типов объектов событий расширяют тип `Event`. Для других новых типов событий, появившихся в стандарте HTML5, их отдельные типы объектов событий не определяются – обработчикам этих событий передается обычный объект `Event`. Данная справочная статья описывает этот «обычный» объект `Event`, плюс некоторые из его подтипов. Свойства в списке ниже, помеченные звездочкой, определяются непосредственно типом `Event`. Эти свойства наследуются такими объектами событий, как `MessageEvent`, и используются при работе с простыми событиями, такими как событие «load» объекта `Window` и событие «playing» объекта `MediaElement`.

Константы

Следующие константы определяют значения свойства `eventPhase`. Это свойство и эти константы не поддерживаются моделью событий, реализованной в IE.

`unsigned short` **`CAPTURING_PHASE = 1`**

Событие посылается перехватывающим обработчикам событий в предках целевого объекта.

`unsigned short` **`AT_TARGET = 2`**

Событие посылается обработчикам целевого объекта

`unsigned short` **`BUBBLING_PHASE = 3`**

Событие всплывает и посылается обработчикам событий в предках целевого объекта.

Свойства

Следующие свойства определяются стандартной моделью для объекта `Event` событий, а также для объектов событий, связанных с событиями мыши и клавиатуры. В этот список также включены свойства, определяемые моделями событий IE и jQuery. Свойства, помеченные звездочкой, определены непосредственно в объекте Event и доступны в любых стандартных объектах событий, независимо от типов событий, с которыми они связаны.

`readonly boolean` **`altKey`**

Указывает, удерживалась ли нажатой клавиша Alt в момент события. Определено для событий мыши и клавиатуры, а также в модели событий IE.

`readonly boolean` **`bubbles*`**

Значение `true`, если тип события поддерживает «всплытие» (и если не вызван метод `stopPropagation()`), и `false` – в противном случае. Отсутствует в модели событий IE.

`readonly unsigned short` **`button`**

Указывает, изменение состояния какой кнопки мыши вызвало событие «mousedown», «mouseup» или «click». Значение 0 соответствует левой кнопке, значение 2 – правой и значение 1 – средней кнопке мыши. Обратите внимание, что значение этого свойства определено только для событий, связанных с изменением состояния кнопки мыши – оно не используется, например, чтобы сообщить, какая кнопка удерживалась при возникновении события «mousemove». Кроме того, это свойство не является битовой маской: оно не позволяет сообщить обработчику об изменении состояния более чем одной кнопки. Наконец, некоторые броузеры генерируют события только для левой кнопки.

Модель событий IE определяет несовместимое свойство `button`. В этом броузере данное свойство является битовой маской: бит 1 устанавливается, когда была нажата левая кнопка, бит 2 – когда была нажата правая кнопка и бит 4 – когда была нажата средняя кнопка (трехкнопочной) мыши. Библиотека jQuery не имитирует стандартное свойство `button` в IE, вместо этого она реализует свойство `which`.

`readonly boolean` **`cancelable*`**

Значение `true`, если действие, предлагаемое по умолчанию и связанное с событием, может быть отменено с помощью метода `preventDefault()`, и `false` – в противном случае. Определено во всех стандартных объектах событий и отсутствует в модели событий IE.

`boolean` **`cancelBubble`**

В модели событий IE, чтобы в обработчике события остановить дальнейшее распространение события вверх по дереву вмещающих объектов, в это свойство нужно записать значение `true`. В стандартной модели событий для этой цели следует использовать метод `stopPropagation()`.

`readonly integer` **`charCode`**

Для событий «keypress» это свойство содержит код Юникода сгенерированного печатного символа. Данное свойство равно нулю в случае нажатия функциональной клавиши. Оно не используется событиями «keydown» и «keyup». Преобразовать это число в строку можно с помощью метода `String.fromCharCode()`. Для событий «keypress» то же самое значение большинство броузеров записывают в свойство `keyCode`. Однако в Firefox свойство `keyCode` не определено для события «keypress», поэтому следует использовать свойство `charCode`. Данное свойство является нестандартным, отсутствует в модели событий IE и не имитируется в модели событий jQuery.

`readonly long` **`clientX`**
`readonly long` **`clientY`**

Координаты X и Y указателя мыши относительно *клиентской области* или окна броузера. Обратите внимание: эти координаты не учитывают величину прокрутки документа; если событие происходит на верхнем краю окна, свойство `clientY` будет равно 0 независимо от того, как далеко выполнена прокрутка документа. Эти свойства определены для всех типов событий мыши, и в модели событий IE, и в стандартной модели. См. также `pageX` и `pageY`.

`readonly boolean` **`ctrlKey`**

Указывает, удерживалась ли нажатой клавиша Ctrl в момент события. Определено для событий мыши и клавиатуры, а также в модели событий IE.

`readonly EventTarget` **`currentTarget*`**

Объект `Element`, `Document` или `Window`, обрабатывающий событие в данный момент. Во время фазы перехвата и всплытия значение свойства отличается от `target`. Отсутствует в модели событий IE, но имитируется в модели событий jQuery.

`readonly DataTransfer` **`dataTransfer`**

Для событий буксировки (drag-and-drop) это свойство определяет объект `DataTransfer`, полностью координирующий выполнение операции буксировки. События буксировки относятся к событиям мыши; любое событие, имеющее это свойство, также будет иметь свойства `clientX`, `clientY` и другие, свойственные событиям мыши. События буксировки «dragstart», «drag» и «dragend» возбуждаются в объекте-источнике; а события «dragenter», «dragover», «dragleave» и «drop» – в объекте-приемнике. Дополнительные сведения об операциях буксировки приводятся в справочной статье `DataTransfer` и в разделе 17.7.

`readonly boolean` **`defaultPrevented*`**

Значение `true`, если обработчик этого события вызвал `defaultPrevented()`, и `false` – в противном случае. Это новое расширение стандартной модели событий, и потому может быть реализовано не всеми броузерами. (Модель событий jQuery определяет метод `isDefaultPrevented()`, действующий подобно этому свойству.)

`readonly long` **`detail`**

Сведения о событии (число). Для событий «click», «mousedown» и «mouseup» это свойство показывает количество щелчков: 1 – одинарный щелчок, 2 – двойной щелчок, 3 – тройной щелчок и т. д. В Firefox это свойство используется событиями «DOMMouseScroll», чтобы сообщить величину прокрутки колесика мыши.

readonly unsigned short **eventPhase***

Текущая фаза распространения события. Значение свойства – одна из трех констант, описанных выше. Не поддерживается в модели событий IE.

readonly boolean **isTrusted***

Значение `true`, если это событие было создано и послано броузером, и `false` – если это искусственное событие, созданное и посланное сценарием на языке JavaScript. Это относительно новое расширение стандартной модели событий, и потому может быть реализовано не всеми броузерами.

readonly Element **fromElement**

Для событий «mouseover» и «mouseout» в модели IE свойство `fromElement` содержит ссылку на объект, с которого двигался указатель мыши. В стандартной модели событий следует использовать свойство `relatedTarget`.

readonly integer **keyCode**

Виртуальный код нажатой клавиши. Это свойство используется всеми типами событий клавиатуры. Код клавиши может зависеть от броузера, операционной системы, самой клавиатуры. Обычно, если на клавише изображен печатный символ, виртуальный код этой клавиши совпадает с кодом символа. Коды функциональных клавиш, не соответствующих печатным символам, могут существенно отличаться, тем не менее множество наиболее часто используемых кодов клавиш можно увидеть в примере 17.8. Это свойство не было стандартизовано, но определяется всеми броузерами, включая IE.

readonly boolean **metaKey**

Признак, показывающий, удерживалась ли клавиша Meta, когда произошло событие. Свойство определено для всех типов событий мыши и клавиатуры, а также в модели событий IE.

readonly integer **offsetX**, **offsetY**

В модели событий IE эти свойства определяют координаты, в которых возникло событие, в координатной системе элемента-источника события (см. описание свойства `srcElement`). Стандартная модель событий не имеет эквивалентных свойств.

readonly integer **pageX**, **pageY**

Эти нестандартные, но широко поддерживаемые свойства подобны свойствам `clientX` и `clientY`, но вместо системы координат окна используют систему координат документа. Эти свойства отсутствуют в модели событий IE, но библиотека jQuery имитирует их во всех броузерах.

readonly EventTarget **relatedTarget***

Ссылается на элемент (обычно элемент документа), который имеет отношение к целевому элементу события. Для событий «mouseover» это элемент, который покинул указатель мыши при наведении на целевой элемент. Для событий «mouseout» это элемент, на который наводится указатель мыши, когда он покидает целевой элемент. Это свойство отсутствует в модели событий IE, но оно имитируется в модели событий jQuery. См. также свойства `fromElement` и `toElement`, определяемые в модели событий IE.

boolean **returnValue**

Чтобы в модели событий IE предотвратить выполнение действия, предусмотренного по умолчанию элементом-источником, в котором возникло событие, данное свойство следует установить в значение `false`. В стандартной модели событий следует использовать метод `preventDefault()`.

`readonly long` **`screenX`**, **`screenY`**

Для событий мыши эти свойства определяют координаты указателя мыши относительно верхнего левого угла экрана. Сами по себе эти свойства не используются, но они определены во всех типах событий мыши и поддерживаются обеими моделями событий, стандартной и IE.

`readonly boolean` **`shiftKey`**

Указывает, удерживалась ли нажатой клавиша Shift в момент события. Определено для событий мыши и клавиатуры, а также в модели событий IE.

`readonly EventTarget` **`srcElement`**

В модели событий IE это свойство определяет объект, в котором было сгенерировано событие. В стандартной модели событий вместо этого свойства следует использовать свойство `target`.

`readonly EventTarget` **`target*`**

Целевой объект события, т. е. объект, в котором было сгенерировано событие. (Все объекты, которые могут быть целевыми объектами событий, реализуют методы интерфейса `EventTarget`.) Это свойство отсутствует в модели событий IE, но имитируется в модели событий jQuery. См. также `srcElement`.

`readonly unsigned long` **`timeStamp*`**

Число, определяющее дату и время, когда произошло событие, или которое можно использовать для определения очередности событий. Многие броузеры возвращают значение времени в секундах, которое можно передать конструктору `Date()`. Однако в Firefox версии 4 и ниже это свойство содержит количество миллисекунд, прошедших с момента включения компьютера. Это свойство не поддерживается в модели событий IE. Модель событий jQuery записывает в это свойство значение в формате, возвращаемом методом `Date.getTime()`.

`Element` **`toElement`**

Для событий «mouseover» и «mouseout» в модели событий IE содержит ссылку на объект, в пределы которого был перемещен указатель мыши. В стандартной модели событий вместо этого свойства следует использовать свойство `relatedTarget`.

`readonly string` **`type*`**

Тип события, которое представляет данный объект `Event`. Это имя, под которым был зарегистрирован обработчик события, или имя свойства обработчика события, без префикса «on». Например, «click», «load» или «submit». Это свойство поддерживаются обеими моделями событий, стандартной и IE.

`readonly Window` **`view`**

Окно (исторически называется «представлением» («view»)), в котором было сгенерировано событие. Это свойство определено для всех стандартных событий пользовательского интерфейса, таких как события мыши и клавиатуры. Не поддерживается в модели событий IE.

`readonly integer` **`wheelDelta`**

Для событий колесика мыши это свойство определяет величину прокрутки по оси Y. Различные броузеры записывают в это свойство разные значения; подробнее об этом рассказывается в разделе 17.6. Это нестандартное свойство, но оно поддерживается всеми броузерами, включая IE версии 8 и ниже.

`readonly integer` **`wheelDeltaX`**
`readonly integer` **`wheelDeltaY`**

В броузерах, поддерживающих мыши с двумя колесиками, эти события определяют величину прокрутки по осям X и Y. Описание, как следует интерпретировать эти свойства, приводится в разделе 17.6. Если определено свойство `wheelDeltaY`, оно будет содержать то же значение, что и свойство `wheelDelta`.

`readonly integer` **`which`**

Это нестандартное, устаревшее свойство поддерживается всеми броузерами, кроме IE, и имитируется в библиотеке jQuery. Для событий мыши это еще одно свойство, аналогичное свойству `button`: значение 1 соответствует левой кнопке, 2 – средней кнопке и 3 – правой. Для событий клавиатуры оно получает то же значение, что и свойство `keyCode`.

Методы

Все следующие методы определены непосредственно в классе `Event`, поэтому все они доступны в любом стандартном объекте `Event`.

`void` **`initEvent(`**`string` *`type`*`, boolean` *`bubbles`*`, boolean` *`cancelable`*`)`

Инициализирует свойства `type`, `bubbles` и `cancelable` объекта `Event`. Создать новый объект события можно вызовом `createEvent()` объекта `Document`, передав ему строку «Event». После инициализации объекта события вызовом этого метода его можно послать любому объекту, поддерживающему интерфейс `EventTarget`, вызвав метод `dispatchEvent()` этого объекта. Другие стандартные свойства объекта события (помимо `type`, `bubbles` и `cancelable`) будут инициализированы во время отправки. Если потребуется создать, инициализировать и послать более сложное искусственное событие, необходимо передать методу `createEvent()` другой аргумент (такой как «MouseEvent») и затем инициализировать полученный объект события с помощью специализированной функции, такой как `initMouseEvent()` (не описывается в этой книге).

`void` **`preventDefault()`**

Сообщает веб-броузеру, чтобы он не выполнял действие по умолчанию для этого события, если таковое предусмотрено. Если событие относится к категории неотменяемых, вызов этого метода не оказывает никакого влияния. Этот метод отсутствует в модели событий IE, но имитируется библиотекой jQuery. В модели событий IE вместо вызова этого метода следует присваивать значение `false` свойству `returnValue`.

`void` **`stopImmediatePropagation()`**

Действует подобно методу `stopPropagation()`, но, кроме того, предотвращает вызов остальных обработчиков, зарегистрированных в этом же элементе документа. Этот метод является новым расширением стандартной модели событий, и потому может быть реализован не во всех броузерах. Не поддерживается моделью событий IE, но имитируется библиотекой jQuery.

`void` **`stopPropagation()`**

Останавливает распространение события по фазам захвата или всплытия и передачу его целевому элементу. Этот метод не отменяет вызов других обработчиков событий того же узла документа, но предотвращает передачу событий любым другим узлам. Не поддерживается моделью событий IE, но имитируется библиотекой jQuery. В IE вместо вызова метода `stopPropagation()` следует присваивать значение `true` свойству `cancelBubble`.

Предлагаемые к реализации свойства

Свойства, перечисленные ниже, предлагаются проектом спецификации «DOM Level 3 Events». Они должны решить основные проблемы несовместимости между броузерами, но пока (на момент написания этих слов) не реализованы ни в одном из броузеров. Если они будут реализованы совместимым способом, это существенно упростит создание переносимого программного кода для обработки событий ввода текста, нажатий клавиш и событий мыши.

readonly unsigned short **buttons**

Напоминает свойство button, поддерживаемое в модели событий IE и описанное выше.

readonly string **char**

Для событий клавиатуры это свойство хранит строку символов (т.е. может содержать более одного символа), сгенерированную событием.

readonly string **data**

Для событий «textinput» определяет введенный текст.

readonly unsigned long **deltaMode**

Для событий колесика мыши это свойство определяет соответствующую интерпретацию свойств deltaX, deltaY и deltaZ. Значением этого свойства может быть одна из констант: DOM_DELTA_PIXEL, DOM_DELTA_LINE, DOM_DELTA_PAGE. Конкретное значение определяется платформой и может зависеть от настроек системы или от факта удерживания нажатыми клавиш-модификаторов во время возникновения события колесика мыши.

readonly long **deltaX**, **deltaY**, **deltaZ**

Для событий колесика мыши эти свойства определяют величину прокрутки по каждой из трех осей.

readonly unsigned long **inputMethod**

Для событий «textinput» это свойство определяет способ ввода текста. Значением этого свойства может быть одна из констант: DOM_INPUT_METHOD_UNKNOWN, DOM_INPUT_METHOD_KEYBOARD, DOM_INPUT_METHOD_PASTE, DOM_INPUT_METHOD_DROP, DOM_INPUT_METHOD_IME, DOM_INPUT_METHOD_OPTION, DOM_INPUT_METHOD_HANDWRITING, DOM_INPUT_METHOD_VOICE, DOM_INPUT_METHOD_MULTIMODAL, DOM_INPUT_METHOD_SCRIPT.

readonly string **key**

Для событий клавиатуры, генерирующих символы, это свойство получает то же значение, что и свойство char. Для событий клавиатуры, не генерирующих символы, это свойство содержит имя нажатой клавиши (такое как, «Tab» или «Down»).

readonly string **locale**

Для событий клавиатуры и событий «textinput» это свойство определяет код языка (например, «en-GB»), идентифицирующий выбранную раскладку клавиатуры, если эта информация доступна.

readonly unsigned long **location**

Для событий клавиатуры это свойство определяет местоположение нажатой клавиши. Значением этого свойства может быть одна из констант: DOM_KEY_LOCATION_STANDARD, DOM_KEY_LOCATION_LEFT, DOM_KEY_LOCATION_RIGHT, DOM_KEY_LOCATION_NUMPAD, DOM_KEY_LOCATION_MOBILE, DOM_KEY_LOCATION_JOYSTICK.

readonly boolean **repeat**

Для событий клавиатуры это свойство будет иметь значение true, если событие вызвано длительным удержанием клавиши в нажатом состоянии, вызвавшем автоповтор ввода.

Предлагаемые к реализации методы

Подобно предлагаемым к реализации свойствам, перечисленным выше, проектом стандарта предлагаются к реализации следующие методы, которые пока не реализованы ни в одном из броузеров.

boolean **getModifierState**(string *modifier*)

Для событий мыши и клавиатуры этот метод возвращает true, если в момент возбуждения события удерживалась нажатой указанная клавиша-модификатор *modifier*, и false – в противном случае. Значением аргумента *modifier* может быть одна из строк: «Alt», «AltGraph», «CapsLock», «Control», «Fn», «Meta», «NumLock», «Scroll», «Shift», «SymbolLock» и «Win».

EventSource

Comet-соединение с HTTP-сервером **EventTarget**

Объект EventSource представляет долгоживущее HTTP-соединение, посредством которого веб-сервер может отправлять клиенту текстовые сообщения. Чтобы использовать события, определяемые стандартом «Server-Sent Events», следует передать URL-адрес сервера конструктору EventSource() и затем зарегистрировать обработчик события «message» в полученном объекте EventSource.

Спецификация «Server-Sent Events» появилась совсем недавно и на момент написания этих строк поддерживалась не во всех броузерах.

Конструктор

new **EventSource**(string *url*)

Создает новый объект EventSource, подключенный к веб-серверу, определяемому аргументом *url*. Адрес *url* интерпретируется относительно URL-адреса документа.

Константы

Следующие константы определяют допустимые значения свойства readyState.

unsigned short **CONNECTING = 0**

Идет установка соединения, или соединение было закрыто и объект EventSource пытается восстановить его.

unsigned short **OPEN = 1**

Соединение установлено и готово к приему событий.

unsigned short **CLOSED = 2**

Соединение было закрыто либо вызовом метода close(), либо в результате фатальной ошибки, не позволяющей восстановить его.

Свойства

readonly unsigned short **readyState**

Состояние соединения. Возможные значения определяются константами, перечисленными выше.

readonly string **url**

Абсолютный URL-адрес, к которому подключен объект EventSource.

Методы

void **close()**

Закрывает соединение. После вызова этого метода объект EventSource не может больше использоваться. Если потребуется вновь установить соединение, следует создать новый объект EventSource.

Обработчики событий

Сетевые взаимодействия выполняются асинхронно, поэтому объект EventSource **возбуждает события после открытия соединения, при появлении ошибок и при получении сообщений от сервера. Обработчики событий можно зарегистрировать с помощью перечисленных далее свойств или воспользовавшись методами интерфейса** EventTarget. **Все события, генерируемые объектом** EventSource, **посылаются самому объекту** EventSource. **Они не всплывают, и для них не предусмотрены действия по умолчанию, которые можно было бы отменить.**

onerror	**Вызывается при обнаружении ошибки. Обработчику передается простой объект** Event.
onmessage	**Вызывается при получении сообщения от сервера. Обработчику передается объект события** MessageEvent, **а текст, отправленный сервером, доступен через свойство** data **этого объекта.**
onopen	**Вызывается при открытии соединения. Обработчику передается простой объект** Event.

EventTarget

объект, способный принимать события

Объекты, для которых генерируются события, и объекты, которые находятся на пути всплывающих событий, должны предоставлять возможность определять обработчики этих событий. Такие объекты обычно определяют свойства обработчиков событий, имена которых начинаются с префикса «on» и, как правило, определяют методы, описываемые ниже. Регистрация обработчиков событий – поразительно сложная тема. За подробностями обращайтесь к разделу 17.2 и обратите внимание, что IE версии 8 и ниже использует другие методы, чем все остальные броузеры; эти методы будут описаны в специальном разделе ниже.

Методы

void **addEventListener**(string *type*, function *listener*, [boolean *useCapture*])

Регистрирует функцию *listener* **в качестве обработчика событий типа** *type*. **Аргумент** *type* **– строка с именем без префикса «on». Аргумент** *useCapture* **должен иметь значение** true, **если регистрируется перехватывающий обработчик (раздел 17.2.3) в предке элемента, являющегося истинной целью события. Обратите внимание, что некоторые броузеры все еще требуют передачи этой функции третьего аргумента, поэтому при регистрации обычного, неперехватывающего, обработчика в третьем аргументе следует передавать** false.

boolean **dispatchEvent**(Event *event*)

Отправляет данному элементу искусственное событие *event*. **Чтобы отправить событие, необходимо создать новый объект** Event **вызовом метода** document.createEvent() **с именем события (таким как «Events», в случае простого события). Затем инициализировать его методом инициализации созданного объекта** Event: **для простых событий – вызовом метода** initEvent() **(см.** Event**). После этого отправить инициализированное событие, передав его рассматриваемому методу. В современных броузерах каждый объект** Event **имеет свойство** isTrusted. **Для искуственных событий, посылаемых программно, это свойство будет иметь значение** false.

Различные типы объектов событий определяют собственные методы инициализации. Эти методы редко используются на практике, имеют длинные списки аргументов и не описываются в этой книге. Если вам потребуется создать, инициали-

зировать и послать искуственное событие, более сложное, чем простой объект `Event`, поищите описание соответствующего метода инициализации в электронной документации.

void **removeEventListener**(string *type*, function *listener*, [boolean *useCapture*])

Удаляет зарегистрированный обработчик *listener* события. Принимает те же аргументы, что и метод `addEventListener()`.

Методы Internet Explorer

IE версии 8 и ниже не поддерживает методы `addEventListener()` и `removeEventListener()`. Вместо них он реализует два более простых метода, которые описываются ниже. (Некоторые важные отличия перечислены в разделе 17.2.4.)

void **attachEvent**(string *type*, function *listener*)

Регистрирует функцию *listener* как обработчик событий типа *type*. Обратите внимание, что этот метод требует, чтобы имя в аргументе *type* включало префикс «on».

void **detachEvent**(string *type*, function *listener*)

Этот метод действует как обратный методу `attachEvent()`.

FieldSet

элемент <fieldset> в HTML-формах **Node, Element, FormControl**

Объект `FieldSet` представляет элемент `<fieldset>` в HTML-формах `<form>`. Объекты `FieldSet` реализуют многие, но не все свойства и методы интерфейса `FormControl`.

Свойства

boolean **disabled**

Значение `true`, если объект `FieldSet` находится в неактивном состоянии. Деактивация элемента `FieldSet` деактивирует содержащиеся в нем элементы форм.

readonly HTMLFormControlsCollection **elements**

Объект, подобный массиву, содержащий все элементы форм, имеющиеся внутри тега `<fieldset>`.

File

файл в локальной файловой системе **Blob**

Тип `File` является подтипом `Blob`. Объект `File` имеет имя и, возможно, дату последнего изменения. Он представляет файл в локальной файловой системе. Получить выбранный пользователем файл можно из массива `files` элемента `<input type=file>` или из массива `files` объекта `DataTransfer`, связанного с объектом `Event`, который получает обработчик события «drop».

Имеется также возможность получить объекты `File`, представляющие файлы в закрытой, изолированной файловой системе, как описано в разделе 22.7. Однако на момент написания этих строк прикладной интерфейс доступа к файловой системе оставался нестабильным, поэтому он не описывается в этом справочнике.

Содержимое файла можно выгрузить на сервер с помощью объекта `FormData` или передав объект `File` методу `XMLHttpRequest.send()`, однако непосредственно с объектом `File` можно выполнить не очень много операций. Чтобы прочитать содержимое файла, следует использовать объект `FileReader` (или любой объект `Blob`).

Свойства

readonly Date **lastModifiedDate**

Дата последнего изменения файла или `null`, если эта информация недоступна.

readonly string **name**

Имя файла (без пути к нему).

FileError

ошибка, возникшая во время чтения файла

Объект `FileError` представляет ошибку, возникшую при чтении файла с помощью объекта `FileReader` или `FileReaderSync`. Если использовался синхронный прикладной интерфейс, возбуждается непосредственно объект `FileError`. При использовании асинхронного прикладного интерфейса объект `FileError` передается обработчику в виде значения свойства `error` объекта `FileReader`.

Обратите внимание, что прикладной интерфейс объекта `FileWriter` (который описывается в разделе 22.7, но пока остается недостаточно стабильным, чтобы его можно было описать в этом справочнике) добавляет в этот объект новые константы с кодами ошибок.

Константы

Ниже перечислены коды ошибок в объекте `FileError`:

unsigned short **NOT_FOUND_ERR = 1**

Файл не существует. (Возможно, был удален после того, как пользователь выбрал его, но перед тем, как программа попыталась прочитать его.)

unsigned short **SECURITY_ERR = 2**

Неопределенная проблема, связанная с безопасностью, вынуждающая броузер запретить вашей программе читать файл.

unsigned short **ABORT_ERR = 3**

Операция чтения файла была прервана.

unsigned short **NOT_READABLE_ERR = 4**

Файл недоступен для чтения, возможно, потому что изменились права доступа к нему или другой процесс заблокировал файл.

unsigned short **ENCODING_ERR = 5**

Вызов `readAsDataURL()` потерпел неудачу, потому что файл оказался слишком длинным, чтобы представить его в виде URL-адреса `data://`.

Свойства

readonly unsigned short **code**

Это свойство определяет тип возникшей ошибки. Это свойство получает значение одной из констант, перечисленных выше.

FileReader

асинхронный интерфейс чтения объекта File или Blob **EventTarget**

Объект `FileReader` определяет асинхронный прикладной интерфейс чтения содержимого объекта `File` или любого объекта `Blob`. Чтобы прочитать файл, следует выполнить следующие действия:

- Создать объект `FileReader` вызовом конструктора `FileReader()`.

- Определить необходимые обработчики событий.
- Передать объект `File` или `Blob` одному из четырех методов чтения.
- Затем, когда будет вызван обработчик `onload`, содержимое файла будет доступно в свойстве `result`. Или, если будет вызван обработчик `onerror`, свойство `error` будет ссылаться на объект `FileError`, содержащий дополнительную информацию.
- После окончания операции чтения при необходимости можно повторно использовать объект `FileReader` или удалить его и создать новый.

Синхронный прикладной интерфейс, который можно использовать в фоновых потоках выполнения, описывается в справочной статье `FileReaderSync`.

Конструктор

`new` **`FileReader()`**

Новый объект `FileReader` создается с помощью конструктора `FileReader()`, который не требует аргументов.

Константы

Следующие константы определяют возможные значения свойства `readyState`:

`unsigned short` **`EMPTY = 0`**

Метод чтения еще не был вызван.

`unsigned short` **`LOADING = 1`**

Выполняется операция чтения.

`unsigned short` **`DONE = 2`**

Операция чтения завершилась, успешно или с ошибкой.

Свойства

`readonly FileError` **`error`**

Если ошибка возникнет во время чтения, это свойство будет ссылаться на объект `FileError`, описывающий ошибку.

`readonly unsigned short` **`readyState`**

Это свойство описывает текущее состояние объекта `FileReader`. Его значением будет одна из трех констант, перечисленных выше.

`readonly any` **`result`**

Если операция чтения завершится успешно, это свойство будет хранить содержимое объекта `File` или `Blob` в виде строки или объекта `ArrayBuffer` (в зависимости от использовавшегося метода чтения). Когда свойство `readyState` имеет значение `LOADING` или когда возбуждается событие «progress», это свойство может хранить неполное содержимое объекта `File` или `Blob`. Если метод чтения еще не был вызван или если возникла ошибка, это свойство будет иметь значение `null`.

Методы

`void` **`abort()`**

Прерывает операцию чтения. Он присваивает свойству `readyState` значение `DONE`, свойству `result` – значение `null` и свойству `error` – объект `FileError` со свойством `code`, установленным в значение `FileError.ABORT_ERR`. После этого возбуждаются события «abort» и «loadend».

`void` **`readAsArrayBuffer`**`(Blob` *`blob`*`)`

- Асинхронно читает данные из объекта *blob* и сохраняет их в свойстве `result` в виде объекта `ArrayBuffer`.

`void` **`readAsBinaryString`**`(Blob` *`blob`*`)`

Асинхронно читает байты данных из объекта *blob*, преобразует их в двоичную строку и сохраняет ее в свойстве `result`. Каждый «символ» в двоичной строке представлен кодом символа в диапазоне от 0 до 255. Извлекать эти значения байтов можно с помощью метода `String.charCodeAt()`. Следует отметить, что двоичные строки являются не самым эффективным представлением двоичных данных: вместо них следует использовать объекты `ArrayBuffer`, когда это возможно.

`void` **`readAsDataURL`**`(Blob` *`blob`*`)`

Асинхронно читает байты данных из объекта *blob*, преобразует их (учитывая тип объекта `Blob`) в URL-адрес `data://` и присваивает полученную строку свойству `result`.

`void` **`readAsText`**`(Blob` *`blob`*`, [string` *`encoding`*`])`

Асинхронно читает байты данных из объекта *blob*, декодирует их с использованием кодировки *encoding* в текстовую строку Юникода и затем присваивает полученную строку свойству `result`. Если аргумент *encoding* не указан, используется кодировка UTF-8 (текст в кодировке UTF-16 определяется и декодируется автоматически, если он начинается с маркера Byte Order Mark).

Обработчики событий

Подобно всем асинхронным прикладным интерфейсам, в своей работе объект `FileReader` опирается на события. Для регистрации обработчиков событий можно использовать свойства, перечисленные ниже, или методы интерфейса `EventTarget`, реализуемого объектом `FileReader`.

События объекта `FileReader` возбуждаются в самом объекте `FileReader`. Они не всплывают, и для них не предусматриваются действия по умолчанию. Обработчикам событий в объекте `FileReader` всегда передается объект `ProgressEvent`. Успешная операция чтения начинается с события «loadstart», за которым следует ноль или более событий «progress», событие «load» и событие «loadend». Неудачная операция чтения начинается с события «loadstart», за которым следует ноль или более событий «progress», событие «error» или «abort» и событие «loadend».

`onabort`

Вызывается, если операция чтения была прервана методом `abort()`.

`onerror`

Вызывается, если возникла какая-либо ошибка. Свойство `error` объекта `FileReader` будет ссылаться на объект `FileError`, который имеет свойство `code` с кодом ошибки.

`onload`

Вызывается в случае успешного завершения операции чтения объекта `File` или `Blob`. Свойство `result` объекта `FileReader` хранит содержимое объекта `File` или `Blob` в виде, зависящем от использовавшегося метода чтения.

`onloadend`

Каждый вызов метода чтения объекта `FileReader` в конечном итоге возбуждает событие «load», «error» или «abort». Кроме того, после каждого из этих событий объект `FileReader` возбуждает событие «loadend» для программ, в которых было бы желательно обрабатывать единственное событие вместо трех.

`onloadstart`

Вызывается после вызова метода чтения, но перед тем как будут прочитаны какие-либо данные.

`onprogress`

Возбуждается примерно 20 раз в секунду, пока выполняется чтение данных из объекта `File` или `Blob`. Объект `ProgressEvent` позволяет узнать количество прочитанных байтов, а свойство `result` объекта `FileReader` может содержать представление этих байтов.

FileReaderSync

синхронный интерфейс чтения объекта File или Blob

Объект `FileReaderSync` является синхронной версией объекта `FileReader`, доступной только в фоновых потоках выполнения, представленных объектами `Worker`. Синхронный интерфейс проще в использовании, чем асинхронный: достаточно просто создать объект вызовом конструктора `FileReaderSync()` и затем вызвать один из его методов чтения, который либо вернет содержимое объекта `File` или `Blob`, либо возбудит объект `FileError`.

Конструктор

`new FileReaderSync()`

Новый объект `FileReaderSync` создается с помощью конструктора `FileReaderSync()`, который не требует аргументов.

Методы

Следующие методы возбуждают объект `FileError`, если операция чтения потерпит неудачу по каким-либо причинам.

`ArrayBuffer` **`readAsArrayBuffer(`**`Blob` *`blob`*`)`

Читает байты из объекта *`blob`* и возвращает их в виде объекта `ArrayBuffer`.

`string` **`readAsBinaryString(`**`Blob` *`blob`*`)`

Читает байты из объекта *`blob`*, преобразует их в двоичную строку (`String.fromCharCode()`) и возвращает ее.

`string` **`readAsDataURL(`**`Blob` *`blob`*`)`

Читает байты из объекта *`blob`*, преобразует их с учетом свойства `type` объекта *`blob`* в URL-адрес `data://` и возвращает его.

`string` **`readAsText(`**`Blob` *`blob`*`, [string` *`encoding`*`])`

Читает байты данных из объекта *`blob`*, декодирует их с использованием кодировки *`encoding`* (или с использованием кодировки UTF-8 или UTF-16, если аргумент *`encoding`* не указан) и возвращает полученную строку.

Form

тег <form> в HTML-документе **Node, Element**

Объект `Form` представляет элемент `<form>` в HTML-документе. Свойство `elements` – это объект `HTMLCollection`, который дает удобный доступ ко всем элементам в форме. Методы `submit()` и `reset()` позволяют программным способом отправлять данные формы или сбрасывать все элементы формы в значения, предлагаемые по умолчанию.

Каждая форма в документе представлена элементом массива `document.forms[]`. Элементы формы (кнопки, поля ввода, переключатели и т. д.) собраны в объекте `Form.elements`, подобном массиву. К именованным элементам форм можно обращаться непосредственно по имени – имя элемента выступает в качестве имени свойства объекта `Form`. Другими словами, обратиться к элементу `Input` со значением «phone» в свойстве `name` в форме `f` можно посредством JavaScript-выражения `f.phone`.

Подробнее об HTML-формах рассказывается в разделе 15.9. Кроме того, дополнительную информацию об элементах форм можно найти в справочных статьях `FormControl`, `FieldSet`, `Input`, `Label`, `Select` и `TextArea`.

Данная справочная статья описывает некоторые особенности форм, определяемые спецификацией HTML5, которые на момент написания этих строк были реализованы не во всех броузерах.

Свойства

Большинство свойств, перечисленных ниже, просто соответствуют HTML-атрибутам с теми же именами.

`string` **`acceptCharset`**

Список из одного или более допустимых кодировок символов, которые могут использоваться для кодирования данных формы при отправке.

`string` **`action`**

URL-адрес, по которому должна быть отправлена форма.

`string` **`autocomplete`**

Строка «on» или «off». Если содержит строку «on», броузер будет предварительно заполнять элементы формы значениями, сохраненными при предыдущем посещении страницы.

`readonly HTMLFormControlsCollection` **`elements`**

Объект, подобный массиву, содержащий элементы формы.

`string` **`enctype`**

Определяет способ кодирования значений элементов формы при отправке. Допустимыми значениями являются:

- «application/x-www-form-urlencoded» (по умолчанию)
- «multipart/form-data»
- «text/plain»

`readonly long` **`length`**

Количество элементов формы, представляемых свойством `elements`. Формы действуют, как если бы они сами были объектами, подобными массивам, содержащими элементы форм, поэтому для формы `f` и целого числа `n` выражение `f[n]` будет эквивалентно выражению `f.elements[n]`.

`string` **`method`**

HTTP-метод отправки формы по URL-адресу в свойстве `action`. Может иметь значение «get» или «post».

`string` **`name`**

Имя формы, определяемое HTML-атрибутом `name`. Значение этого свойства можно использовать в качестве имени свойства объекта документа, значением которого будет данный объект `Form`.

`boolean` **`noValidate`**

Значение `true`, если форма не должна проверяться перед отправкой. Соответствует HTML-атрибуту `novalidate`.

`string` **`target`**

Имя окна или фрейма, где должен отображаться документ, возвращаемый в ответ на выполнение операции отправки формы.

Методы

`boolean` **`checkValidity()`**

В броузерах, поддерживающих возможность проверки форм, этот метод проверяет корректность введенных данных в каждом элементе формы. Он возвращает `true`, если все данные корректны. Если какой-либо элемент управления содержит недопустимые данные, он возбуждает событие «invalid» в этом элементе формы и возвращает `false`.

`void` **`dispatchFormChange()`**

Возбуждает событие «formchange» в каждом элементе данной формы. Обычно это делается автоматически, когда действия пользователя приводят к возбуждению события «change», поэтому вызывать этот метод обычно не требуется.

`void` **`dispatchFormInput()`**

Возбуждает событие «forminput» в каждом элементе данной формы. Обычно это делается автоматически, когда действия пользователя приводят к возбуждению события «input», поэтому вызывать этот метод обычно не требуется.

`void` **`reset()`**

Сбрасывает все элементы ввода формы к их значениям по умолчанию.

`void` **`submit()`**

Выполняет отправку формы вручную, не возбуждая событие «submit».

Обработчики событий

Следующие обработчики событий, связанные с формами, определены в объекте `Element`, но описываются здесь, потому что возбуждаются в элементах `Form`.

`onreset`

Вызывается непосредственно перед тем, как форма будет сброшена в исходное состояние. Чтобы предотвратить сброс, обработчик может вернуть `false` или отменить событие.

`onsubmit`

Вызывается непосредственно перед отправкой формы. Чтобы предотвратить отправку, обработчик может вернуть `false` или отменить событие.

FormControl

общие особенности всех элементов форм

Большинство элементов HTML-форм являются элементами `<input>`, но формы могут также содержать элементы `<button>`, `<select>` и `<textarea>`. Данная справочная статья описывает общие особенности всех этих элементов. Введение в HTML-формы приводится в разделе 15.9, а дополнительную информацию о формах и об элементах форм можно найти в справочных статьях `Form`, `Input`, `Select` и `TextArea`.

Элементы `<fieldset>` и `<output>` реализуют большинство, но не все, свойства, описываемые здесь. Данный справочник классифицирует объекты `FieldSet` и `Output` как подтипы объекта `FormControl`, хотя они реализуют не все свойства.

Данная справочная статья описывает некоторые особенности форм (такие как проверка данных формы), введенные спецификацией HTML5, которые на момент написания этих строк были реализованы не во всех броузерах.

Свойства

`boolean` **`autofocus`**

Значение `true`, если элемент должен автоматически получать фокус ввода сразу после загрузки документа. (Элементы `FieldSet` и `Output` не реализуют это свойство.)

`boolean` **`disabled`**

Значение `true`, если элемент формы находится в неактивном состоянии. Неактивные элементы не откликаются на ввод пользователя и не подвергаются проверке. (Элементы `Output` не реализуют это свойство; элементы `FieldSet` используют его для управления активностью всех элементов, содержащихся в них.)

`readonly Form` **`form`**

Ссылка на элемент `Form`, который является владельцем данного элемента, или `null`, если таковой отсутствует. Если элемент формы находится внутри элемента `<form>`, эта форма является его владельцем. В противном случае, если элемент формы имеет HTML-атрибут `form`, определяющий значение атрибута `id` элемента `<form>`, владельцем элемента будет указанная форма.

`readonly NodeList` **`labels`**

Объект, подобный массиву, содержащий элементы `Label`, связанные с элементами этой формы. (Объекты `FieldSet` не реализуют это свойство.)

`string` **`name`**

Значение HTML-атрибута `name` для данного элемента формы. Имена элементов форм можно использовать в качестве имен свойств элемента `Form`: значениями таких свойств являются элементы форм. Имена элементов форм также можно использовать для идентификации данных при отправке формы.

`string` **`type`**

Для элементов `<input>` свойство `type` имеет значение атрибута `type` или значение «text», если атрибут `type` не указан в теге `<input>`. Для элементов `<button>`, `<select>` и `<textarea>` свойство `type` имеет значение «button», «select-one» (или «select-multiple», если установлен атрибут `multiple`) и «textarea», соответственно. Для элементов `<fieldset>` свойство `type` имеет значение «fieldset», а для элементов `<output>` – значение «output».

`readonly string` **`validationMessage`**

Если элемент формы содержит допустимые данные или не подвергается проверке, это свойство будет содержать пустую строку. Иначе это свойство будет содержать локализованную строку, описывающую причину, по которой введенные данные признаны некорректными.

`readonly FormValidity` **`validity`**

Это свойство ссылается на объект, который определяет корректность даных в этом элементе формы и описывает причину, если данные признаны некорректными.

`string` **`value`**

Каждый элемент формы имеет строковое свойство `value`, которое используется при отправке формы. Для текстовых элементов форм значением этого свойства является текст, введенный пользователем. Для кнопок – значение HTML-атрибута `value`. Для элементов `Output` это свойство подобно свойству `textContent`, унаследованному от объекта `Node`. Элементы `FieldSet` не реализуют это свойство.

`readonly boolean` **`willValidate`**

Это свойство имеет значение `true`, если элемент формы подвергается проверке, и `false` – в противном случае.

Обработчики событий

Элементы форм определяют следующие свойства обработчиков событий. Обработчики можно также регистрировать с помощью методов интерфейса `EventTarget`, который реализуют все элементы:

Обработчик событий	Когда вызывается
`onformchange`	Когда в любом элементе формы возбуждается событие «change», форма рассылает невсплывающее событие «formchange» всем своим элементам. Элементы форм могут использовать это свойство для определения факта изменений в соседних элементах формы.
`onforminput`	Когда в любом элементе формы возбуждается событие «input», форма рассылает невсплывающее событие «forminput» всем своим элементам. Элементы форм могут использовать это свойство для определения факта изменений в соседних элементах формы.
`oninvalid`	Если в ходе проверки выяснится, что элемент формы содержит некорректные данные, в нем будет возбуждено событие «invalid». Это событие не всплывает, но если его отменить, броузер не выведет сообщение об ошибке для этого элемента.

Методы

`boolean` **`checkValidity()`**

Возвращает `true`, если элемент формы содержит корректные данные (или если этот элемент не подвергается проверке). Иначе возбуждает событие «invalid» в данном элементе и возвращает `false`.

`void` **`setCustomValidity`**`(string` *`error`*`)`

Если в аргументе *`error`* передать непустую строку, этот метод пометит данный элемент формы как содержащий недопустимые данные и будет использовать аргумент *`error`* как локализованное сообщение, чтобы известить пользователя о причинах. Если передать в аргументе *`error`* пустую строку, все предыдущие строки *`error`* будут удалены, а объект будет помечен как содержащий допустимые данные.

FormData

тело HTTP-запроса multipart/form-data

Тип `FormData` является частью спецификации «XMLHttpRequest Level 2» (XHR2), которая упрощает отправку данных в формате «multipart/form-data» в виде HTTP-запросов PUT с помощью объекта `XMLHttpRequest`. Использование этого формата необходимо, например, когда в одном запросе требуется выгрузить несколько объектов `File`.

Создайте объект `FormData` с помощью конструктора и затем добавьте в него пары имя/значение с помощью метода `append()`. После того как будут добавлены все части, составляющие тело запроса, объект `FormData` можно передать методу `send()` объекта `XMLHttpRequest`.

Конструктор

`new` **`FormData()`**

Этот конструктор, не имеющий аргументов, возвращает пустой объект `FormData`.

Методы

`void` **`append`**`(string` *`name`*`, any` *`value`*`)`

Добавляет в объект `FormData` новую часть с именем *`name`* и значением *`value`*. Аргумент *`value`* может быть строкой или объектом `Blob` (напомню, что тип `File` является подтипом `Blob`).

FormValidity

реализует проверку элемента формы

Свойство `validity` объекта `FormControl` ссылается на объект `FormValidity`, который является представлением признака корректности данных в этом элементе формы. Если свойство `valid` имеет значение `false`, элемент формы содержит недопустимые данные и по крайней мере одно из других свойств, определяющих природу ошибки (или ошибок), будет иметь значение `true`.

Проверка форм является особенностью, введенной спецификацией HTML5, которая на момент написания этих строк была реализована не во всех броузерах.

Свойства

`readonly boolean` **`customError`**

Сценарий вызвал метод `FormControl.setCustomValidity()` данного элемента.

`readonly boolean` **`patternMismatch`**

Введенные данные не соответствуют регулярному выражению.

`readonly boolean` **`rangeOverflow`**

Объем введенных данных слишком велик.

`readonly boolean` **`rangeUnderflow`**

Объем введенных данных слишком мал.

`readonly boolean` **`stepMismatch`**

Введенные данные не соответствуют указанному шагу.

`readonly boolean` **`tooLong`**

Объем введенных данных слишком велик.

`readonly boolean` **`typeMismatch`**

Введенные данные имеют неверный тип.

`readonly boolean` **`valid`**

Если это свойство имеет значение `true`, элемент формы содержит корректные данные и все другие свойства имеют значение `false`. Если это свойство имеет значение `false`, элемент формы содержит недопустимые данные и, по крайней мере, одно из других свойств имеет значение `true`.

`readonly boolean` **`valueMissing`**

Элемент формы пуст, хотя он должен быть заполнен.

Geocoordinates

географическое местонахождение

Объект этого типа является представлением точки на поверхности Земли.

Свойства

readonly double **accuracy**

Точность определения широты и долготы (свойства latitude и longitude) в метрах.

readonly double **altitude**

Высота над уровнем моря в метрах или null, если информация о высоте недоступна.

readonly double **altitudeAccuracy**

Точность определения высоты (свойство altitude) над уровнем моря в метрах. Если свойство altitude имеет значение null, свойство altitudeAccuracy также будет иметь значение null.

readonly double **heading**

Направление движения пользователя в градусах относительно направления на истинный север или null, если информация о направлении недоступна. Если информация о высоте доступна, но скорость (свойство speed) движения равна нулю, то свойство heading будет иметь значение NaN.

readonly double **latitude**

Широта местоположения пользователя в градусах с долями к северу от экватора.

readonly double **longitude**

Долгота местоположения пользователя в градусах с долями к востоку от Гринвичского меридиана.

readonly double **speed**

Скорость движения пользователя в метрах в секунду или null, если информация о скорости недоступна. Это свойство никогда не принимает отрицательные значения. См. также heading.

Geolocation

позволяет получить широту и долготу местоположения пользователя

Объект Geolocation определяет методы, позволяющие получить точные географические координаты местоположения пользователя. В броузерах, поддерживающих такую возможность, объект Geolocation можно получить через объекта Navigator, обратившись к свойству navigator.geolocation. Методы, описываемые ниже, опираются на использование некоторых других типов: местоположение определяется в форме объекта Geoposition, а ошибки – в форме объектов GeolocationError.

Методы

void **clearWatch**(long *watchId*)

Останавливает слежение за местонахождением пользователя. В аргументе *watchId* должно передаваться значение, полученное соответствующим вызовом метода watchPosition().

void **getCurrentPosition**(function *success*, [function *error*], [object *options*])

Асинхронно определяет местонахождение пользователя с учетом параметров *options* (перечень свойств объекта *option* приводится ниже). Этот метод немедленно возвращает управление, а когда местонахождение пользователя будет определено, указанной функции обратного вызова *success* будет передан объект `Geoposition`. Или в случае ошибки (возможно из-за того, что пользователь не дал разрешение на определение его координат) функции обратного вызова *error* будет передан объект `GeolocationError`.

`long` **`watchPosition`**`(function` *`success`*`, [function` *`error`*`], [object` *`options`*`])`

Этот метод похож на метод `getCurrentPosition()`, но после определения текущего местонахождения пользователя он продолжает следить за его координатами и вызывает функцию *success* каждый раз, когда обнаружит существенное их изменение. Возвращает число, которое можно передать методу `clearWatch()`, чтобы остановить слежение за местонахождением пользователя.

Параметры

Аргумент *options*, передаваемый методам `getCurrentPosition()` и `watchPosition()`, является обычным объектом, содержащим ноль или более свойств из числа следующих:

`boolean` **`enableHighAccuracy`**

Этот параметр говорит о желательности определения координат с высокой точностью, даже если это повлечет увеличение расхода энергии в аккумуляторах. По умолчанию имеет значение `false`. В устройствах, способных определять местонахождение посредством измерения мощности сигналов WiFi или с помощью GPS, установка этого параметра в значение `true` обычно означает «использовать GPS».

`long` **`maximumAge`**

Этот параметр определяет максимальное время (в миллисекундах), прошедшее с того момента, как объект `Geoposition` был передан функции обратного вызова *success*. По умолчанию имеет значение 0, т.е. каждый вызов метода `getCurrentPosition()` или `watchPosition()` будет заново определять местонахождение. Если установить этот параметр в значение 60000, например, то реализации будет позволено возвращать любой объект `Geoposition`, полученный в течение последней минуты.

`long` **`timeout`**

Этот параметр определяет продолжительность ожидания выполнения запроса в миллисекундах. По умолчанию имеет значение `Infinity`. По истечении указанного интервала времени будет вызвана функция обратного вызова *error*. Обратите внимание, что время ожидания разрешения пользователя на получение его местонахождения не входит в это значение параметра `timeout`.

GeolocationError

ошибка, возникшая в ходе определения местонахождения пользователя

Если попытка определить географическое местонахождение пользователя окончилась неудачей, будет вызвана функция обратного вызова *error* с объектом `GeolocationError`, описывающим ошибку.

Константы

Следующие константы являются возможными значениями свойства `code`:

`unsigned short` **`PERMISSION_DENIED = 1`**

Пользователь не дал разрешение на определение его местонахождения.

`unsigned short` **`POSITION_UNAVAILABLE = 2`**

Местонахождение не может быть определено по невыясненным причинам. Это может быть вызвано, например, сетевой ошибкой.

`unsigned short` **`TIMEOUT = 3`**

Местонахождение не может быть определено в течение установленного интервала времени (см. описание параметра `timeout` в справочной статье `Geolocation`).

Свойства

`readonly unsigned short` **`code`**

Это свойство может иметь одно из трех значений, описанных выше.

`readonly string` **`message`**

Текст сообщения, более детально описывающего ошибку. Сообщение предназначено для отладки и не подходит для уведомления конечного пользователя.

Geoposition

информация о местонахождении с указанием времени определения

Объект `Geoposition` представляет географическое местонахождение пользователя в указанный момент времени. Объекты этого типа имеют всего два свойства: время и ссылку на объект `Geocoordinates`, хранящий фактические значения координат.

Свойства

`readonly Geocoordinates` **`coords`**

Это свойство ссылается на объект `Geocoordinates`, свойства которого определяют широту, долготу и другие параметры местонахождения пользователя.

`readonly unsigned long` **`timestamp`**

Время в миллисекундах с начала эпохи, когда были определены эти координаты. При необходимости на основе этого значения можно создать объект `Date`.

HashChangeEvent

объект события, поставляемый по событию «hashchange» Event

Событие «hashchange» возбуждается броузером, когда изменяется идентификатор фрагмента (часть URL-адреса, начинающаяся с символа решетки #) документа. Это может происходить вследствие изменения свойства `hash` объекта `Location` или при перемещении по истории посещений щелчком на кнопке броузера Back (Назад) или Forward (Вперед). В любом из этих случаев броузер генерирует событие «hashchange». Обработчику этого события передается объект `HashChangeEvent`. Подробное обсуждение механизма управления историей посещения, а также свойства `location.hash` и события «hashchange» можно найти в разделе 22.2.

Свойства

`readonly string` **`newURL`**

Это свойство хранит новое значение свойства `location.href`. Обратите внимание, что это полный URL-адрес, а не только идентификатор фрагмента.

`readonly string` **`oldURL`**

Это свойство хранит прежнее значение свойства `location.href`.

History

журнал посещений объекта Window

Объект `History` представляет историю посещений окна. Однако по соображениям безопасности объект `History` не позволяет получать из сценариев доступ к хранящимся в нем URL-адресам. Методы объекта History позволяют сценариям лишь перемещаться вперед и назад по истории посещений и добавлять в нее новые записи.

Свойства

`readonly long` **`length`**

Это свойство определяет количество URL-адресов в журнале (истории) посещений броузера. Знание размера этого списка не особенно полезно, поскольку нет способа определить индекс текущего отображаемого документа в этом списке.

Методы

`void` **`back()`**

В результате вызова метода `back()` окно или фрейм, которому принадлежит объект `History`, заново открывает URL-адрес (если он есть), открытый непосредственно перед текущим. Вызов этого метода имеет тот же эффект, что и щелчок на кнопке Back броузера. Он также эквивалентен инструкции:

```
history.go(-1);
```

`void` **`forward()`**

В результате вызова метода `forward()` окно или фрейм, которому принадлежит объект `History`, заново открывает URL-адрес (если он есть), открытый непосредственно после текущего. Вызов этого метода имеет тот же эффект, что и щелчок на кнопке Forward броузера. Он также эквивалентен инструкции:

```
history.go(1);
```

`void` **`go(`**`[long` *`delta`*`]`**`)`**

Метод `History.go()` принимает целочисленный аргумент и вынуждает броузер открыть URL-адрес, отстоящий от текущего в журнале истории посещений на указанное число позиций. Положительное значение соответствует переходу вперед по истории посещений, а отрицательное – переходу назад. То есть вызов `history.go(-1)` эквивалентен вызову `history.back()` и имеет тот же эффект, что и щелчок на кнопке Back. При вызове с аргументом 0 или вообще без аргумента этот метод перезагрузит текущий документ.

`void` **`pushState(`**`any` *`data`*`, string` *`title`*`, [string` *`url`*`]`**`)`**

Добавляет новую запись в журнал посещений для данного окна, сохраняя структурированную копию (см. врезку «Структурированные копии» в главе 22) данных *data*, а также значения *title* и *url*. Если позднее пользователь воспользуется механизмом истории посещений броузера, чтобы вернуться к этому сохраненному состоянию, в окне будет сгенерировано событие «popstate» и обработчику будет передан объект `PopStateEvent` с еще одной структурированной копией значения аргумента *data* в его свойстве `state`.

Аргумент *title* определяет имя сохраненного состояния, и броузеры могут отображать его в графическом интерфейсе управления историей посещений. (На момент написания этих строк броузеры игнорировали данный аргумент). Если указан аргумент *url*, он будет отображаться в строке ввода адреса и обеспечит сохранение информации о состоянии, благодаря чему его можно будет использовать для создания закладки или передачи другим пользователям. Аргумент *url* интерпретируется относительно текущего значения `document location`. Если в аргументе *url* указан абсолютный URL-адрес, он должен иметь то же происхождение, что и текущий документ. Чаще всего URL-адреса используются на практике для изменения идентификатора фрагмента документа, начинающегося с символа `#`.

`void` **`replaceState`**`(any` *`data`*`, string` *`title`*`, [string` *`url`*`])`

Этот метод похож на метод `pushState()`, за исключением того, что вместо создания новой записи в истории посещений окна он изменяет текущую запись, сохраняя в ней новые значения *data*, *title* и *url*.

HTMLCollection

коллекция HTML-элементов, доступных по позиции или по имени

Объект `HTMLCollection` – это подобный массиву объект, доступный только для чтения, содержащий объекты `Element` и определяющий свойства, соответствующие значениям атрибутов `name` и `id` хранящихся в нем элементов. Объект `Document` определяет свойства типа `HTMLCollection`, такие как `forms` и `image`.

Объекты `HTMLCollection` определяют методы `item()` и `namedItem()` для извлечения элементов по номеру позиции или имени, но они практически не используются на практике: объект `HTMLCollection` можно рассматривать как обычный объект и обращаться к его свойствам и элементам массива. Например:

```
document.images[0]     // По индексу элемента в коллекции HTMLCollection
document.forms.address // По имени элемента в коллекции HTMLCollection
```

Свойства

`readonly unsigned long` **`length`**

Количество элементов в коллекции.

Методы

`Element` **`item`**`(unsigned long` *`index`*`)`

Возвращает элемент коллекции, расположенный в позиции *index*, или `null`, если индекс *index* выходит за границы массива. Этот метод можно не вызывать явно, а указать индекс в квадратных скобках (как для массива).

`object` **`namedItem`**`(string` *`name`*`)`

Возвращает первый элемент из коллекции, имеющий значение *name* в атрибуте `id` или `name` либо `null`, если такой элемент отсутствует. Этот метод можно не вызывать явно, а указать имя в квадратных скобках.

HTMLDocument

см. Document

HTMLElement

см. Element

HTMLFormControlsCollection

объект, подобный массиву, содержащий элементы форм **HTMLCollection**

Объект `HTMLFormControlsCollection` является специализированным подтипом `HTMLCollection`, используемым элементами `Form` для представления коллекций элементов форм. Подобно объекту `HTMLCollection`, его можно использовать как массив, используя числовые индексы, или как объект, индексируя его значениями атрибутов `name` или `id` элементов форм. HTML-формы часто включают несколько элементов (обычно радиокнопок или флажков), имеющих одинаковые значения в атрибуте `name`, и объект `HTMLFormControlsCollection` обрабатывает их иначе, чем обычный оъект `HTMLCollection`.

При обращении к свойству объекта `HTMLFormControlsCollection`, которому соответствуют несколько элементов с одинаковыми именами, объект `HTMLFormControlsCollection` возвращает объект, подобный массиву, содержащий все элементы формы, использующие это имя. Кроме того, возвращаемый объект, подобный массиву, имеет свойство `value`, которое содержит значение атрибута `value` первой отмеченной радиокнопки с этим именем. Этому свойству можно даже присвоить значение, чтобы отметить соответствующую ему радиокнопку.

HTMLOptionsCollection

коллекция элементов Option **HTMLCollection**

Объект `HTMLOptionsCollection` является специализированным подтипом `HTMLCollection`, который представляет элементы `Option`, имеющиеся в элементе `Select`. Он переопределяет метод `namedItem()`, чтобы обеспечить возможность работы с несколькими элементами `Option`, имеющими одинаковые имена, и определяет методы добавления и удаления элементов. По историческим причинам объект `HTMLOptionsCollection` определяет доступное для записи свойство `length`, которое можно использовать для усечения или расширения коллекции.

Свойства

`unsigned long` **`length`**

Возвращает количество элементов в коллекции. Однако, в отличие от свойства `length` обычного объекта `HTMLCollection`, это свойство доступно не только для чтения. Если присвоить ему значение меньше текущего, коллекция элементов `Option` будет усечена, а элементы `Option`, оказавшиеся за пределами коллекции, будут удалены из элемента `Select`. Если присвоить свойству `length` значение больше текущего, будут созданы и добавлены в элемент `Select` и в коллекцию новые пустые элементы `<option/>`.

`long` **`selectedIndex`**

Индекс первого выбранного элемента `Option` в коллекции или -1, если ни один элемент `Option` не был выбран. Это свойство можно использовать, чтобы программно выбрать требуемый элемент.

Методы

`void` **`add`**`(Element` *`option`*`, [any` *`before`*`])`

Вставляет элемент *`option`* (который должен быть элементом `<option>` или `<optgroup>`) в данную коллекцию (и в элемент `Select`), в позицию, определяемую аргументом *`before`*. Если аргумент *`before`* имеет значение `null`, новый элемент вставляется в конец коллекции. Если аргумент *`before`* имеет целочисленное значение, новый элемент будет вставлен перед элементом, который в текущий момент имеет этот индекс. Если передать в аргументе *`before`* другой элемент `Element`, *`option`* будет вставлен перед этим элементом.

`Element` **`item`**`(unsigned long` *`index`*`)`

Объект `HTMLOptionsCollection` наследует этот метод от `HTMLCollection`. Он возвращает элемент с индексом *`index`* или `null`, если индекс выходит за границы коллекции.

Коллекцию можно также индексировать непосредственно, используя квадратные скобки и не вызывая этот метод явно.

`object` **namedItem**(`string` *name*)

Возвращает все элементы `Option` из коллекции, имеющие значение *name* в атрибуте `id` или `name`. Если элементы с таким именем отсутствуют, возвращается значение `null`. Этот метод можно не вызывать явно, а указать имя в квадратных скобках. Если заданному имени соответствует только один элемент `Option`, возвращается этот элемент. Если заданному имени соответствует более одного элемента, возвращается объект `NodeList` с этими элементами. Обратите внимание, что объекты `HTMLOptionsCollection` можно индексировать непосредственно, используя значение *name* как имя свойства, вместо явного вызова этого метода.

`void` **remove**(`long` *index*)

Удаляет из коллекции элемент `<option>` с индексом *index*. При вызове без аргумента или со значением аргумента, которое выходит за границы коллекции, может удалить первый элемент коллекции.

IFrame

HTML-тег <iframe> **Node, Element**

Объект `IFrame` представляет элемент `<iframe>` в HTML-документе. Если попробовать отыскать элемент `<iframe>` с помощью метода `getElementById()` или подобного ему, вы получите объект `IFrame`. Однако, если обратиться к элементу `<iframe>` через свойство `frames` объекта `Window` или используя имя элемента `<iframe>` как свойство содержащего его окна, вы получите объект `Window`, представляющий элемент `<iframe>`.

Свойства

`readonly Document` **contentDocument**

Документ, содержащий данный элемент `<iframe>`. Если документ отображается в `<iframe>` с другим происхождением, доступ к этому документу будет закрыт из-за ограничений, накладываемых политикой общего происхождения (раздел 13.6.2).

`readonly Window` **contentWindow**

Объект `Window`, содержащий элемент `<iframe>`. (Свойство `frameElement` этого объекта `Window` будет ссылаться обратно на данный объект `IFrame`.)

`string` **height**

Высота элемента `<iframe>` в CSS-пикселах. Это свойство соответствует атрибуту `height`.

`string` **name**

Имя элемента `<iframe>`. Это свойство соответствует атрибуту `name`, а его значение можно присваивать свойству `target` объектов `Link` и `Form`.

`readonly DOMSettableTokenList` **sandbox**

Это свойство соответствует HTML5-атрибуту `sandbox` и может использоваться как строка или как множество отдельных лексем.

Атрибут `sandbox` определяет, какие дополнительные ограничения должны накладываться броузером на содержимое, отображаемое в элементе `<iframe>`. Если атрибут `sandbox` присутствует в элементе, но имеет пустое значение, содержимое фрейма `<iframe>` будет интерпретироваться как имеющее другое происхождение, и ему не будет позволено запускать сценарии, отображать формы и изменять свойство `location` окна, содержащего фрейм. Атрибуту `sandbox` можно также присвоить список лексем, разделенных пробелами, снимающих эти дополнительные ограничения. Допустимыми лексемами являются: «allow-same-origin», «allow-scripts», «allow-forms» и «allow-top-navigation».

На момент написания этих строк атрибут `sandbox` был реализован не во всех броузерах. Дополнительные сведения приводятся в справочной статье `HTML`.

`boolean` **`seamless`**

Это свойство соответствует атрибуту `seamless`. Если оно имеет значение `true`, броузер должен отображать содержимое элемента `<iframe>` так, как если бы оно было составной частью объемлющего документа. Отчасти это означает, что броузер должен применить к содержимому фрейма стили CSS вмещающего документа.

Атрибут `seamless` был введен в спецификации HTML5 и на момент написания этих строк был реализован не во всех броузерах.

`string` **`src`**

Это свойство соответствует атрибуту `src` элемента `<iframe>`: он определяет URL-адрес содержимого фрейма.

`string` **`srcdoc`**

Это свойство соответствует атрибуту `srcdoc` и определяет содержимое элемента `<iframe>` в виде строки. Атрибут `srcdoc` был введен в спецификации HTML5 совсем недавно и на момент написания этих строк был реализован не во всех броузерах.

`string` **`width`**

Ширина фрейма в CSS-пикселах. Это свойство соответствует атрибуту `width`.

Image

тег <img> в HTML-документе **Node, Element**

Объекты Image встраиваются в HTML-документ в виде тегов `<img>`. Изображения, присутствующие в документе, собираются в виде массива `document.images[]`.

Свойство `src` объекта `Image` представляет наибольший интерес. Когда вы устанавливаете это свойство, броузер загружает и показывает изображение, заданное новым значением. Это позволяет создавать такие эффекты, как смена изображений и анимация. Соответствующие примеры приводятся в разделе 21.1.

Имеется возможность создавать невидимые объекты `Image`, добавляя новые элементы `<img>` вызовом метода `document.createElement()` или конструктора `Image()`. Обратите внимание: этот конструктор не имеет аргумента, который определял бы изображение для загрузки; чтобы загрузить изображение, достаточно просто установить свойство `src` созданного вами объекта `Image`. Чтобы фактически отобразить изображение, объект `Image` необходимо вставить в документ.

Конструктор

`new` **`Image`**`([unsigned long` *`width`*`, unsigned long` *`height`*`])`

Как и любой другой HTML-элемент, новый объект `Image` можно создать с помощью метода `document.createElement()`. Однако по историческим причинам клиентский JavaScript также определяет конструктор `Image()`, позволяющий сделать то же самое. Если указаны аргументы *`width`* и/или *`height`*, их значения будут присвоены атрибутам `width` и `height` тега `<img>`.

Свойства

Помимо свойств, перечисленных ниже, элементы `Image` также предоставляют свойства, соответствующие HTML-атрибутам `alt`, `usemap`, `ismap`.

`readonly boolean` **`complete`**

Значение `true`, если свойство `src` не было определено или изображение было загружено полностью, в противном случае – `false`.

`unsigned long` **`height`**

Высота области на экране в CSS-пикселах, в которой отображается данное изображение. Изменение значения этого свойства приводит к изменению высоты изображения на экране.

`readonly unsigned long` **`naturalHeight`**

Высота самого изображения.

`readonly unsigned long` **`naturalWidth`**

Ширина самого изображения.

`string` **`src`**

URL-адрес изображения. Присваивание значения этому свойству вынуждает броузер загрузить указанное изображение. Если объект `Image` был вставлен в документ, он отобразит новое изображение.

`unsigned long` **`width`**

Ширина области на экране в CSS-пикселах, в которой отображается данное изображение. Изменение значения этого свойства приводит к изменению ширины изображения на экране.

ImageData

массив пикселов в элементе <canvas>

Объект `ImageData` хранит красную, зеленую и синюю составляющие, а также уровень прозрачности для каждого пиксела в прямоугольной области. Получить объект `ImageData` можно с помощью метода `createImageData()` или `getImageData()` объекта `CanvasRenderingContext2D` из тега `<canvas>`.

Свойства `width` и `height` определяют размеры прямоугольника в пикселах. Свойство `data` – это массив, хранящий информацию о пикселах. Пикселы размещаются в массиве `data[]` в направлении слева направо и сверху вниз. Каждый пиксел состоит из четырех байт, представляющих компоненты R, G, B и A, именно в этом порядке. Таким образом, получить доступ к компонентам цвета пиксела с координатами `(x,y)` внутри объекта `ImageData` можно так:

```
var offset = (x + y*image.width) * 4;
var red = image.data[offset];
var green = image.data[offset+1];
var blue = image.data[offset+2];
var alpha = image.data[offset+3];
```

Массив `data[]` не является истинным массивом – это объект, подобный массиву, оптимизированный для хранения целочисленных элементов со значениями в диапазоне от 0 до 255. Элементы массива доступны для чтения и записи, но свойство `length` массива имеет фиксированное значение. Для любого объекта `i` типа `ImageData` значение свойства `i.data.length` всегда будет равно значению выражения `i.width * i.height * 4`.

Свойства

`readonly byte[]` **`data`**

Ссылка, доступная только для чтения, на объект, подобный массиву, доступный для чтения и записи, элементами которого являются байты.

`readonly unsigned long` **`height`**

Количество строк пикселов изображения в массиве `data`.

`readonly unsigned long` **`width`**

Количество пикселов в строке изображения, в массиве `data`.

Input

HTML-элемент <input> **Node, Element, FormControl**

Объект `Input` представляет HTML-тег `<input>`. Его внешний вид и поведение определяется атрибутом `type`: элемент `Input` может представлять, например, простое текстовое поле ввода, флажок, радиокнопку, простую кнопку или элемент выбора файла. Так как элемент `<input>` может представлять самые разные элементы форм, объект `Input` является одним из самых сложных. Краткий обзор HTML-форм и их элементов приводится в разделе 15.9. Обратите внимание, что некоторые из важных свойств объекта `Input` (такие как `type`, `value`, `name` и `form`) описываются в справочной статье `FormControl`.

Свойства

Помимо свойств, перечисленных ниже, объекты `Input` также поддерживают все свойства, определенные в объектах `Element` и `FormControl`. Свойства в этом списке, помеченные звездочкой, являются новыми, введенными спецификацией HTML5, и на момент написания этих строк они были реализованы не во всех броузерах.

`string` **`accept`**

Если свойство `type` имеет значение «file», это свойство содержит список MIME-типов, разделенных запятыми, определяющих типы файлов, которые могут быть выбраны. Допустимыми являются также строки «audio/*», «video/*» и «image/*». Соответствует атрибуту `accept`.

`string` **`autocomplete`**

Значение `true`, если броузеру разрешено предварительно заполнять этот элемент `Input` значением, сохранившимся с предыдущего сеанса. Соответствует атрибуту `autocomplete`. См. также описание свойства `autocomplete` объекта `Form`.

`boolean` **`checked`**

Для радиокнопок и флажков данное свойство указывает, является соответствующий элемент «отмеченным» или нет. Изменение этого свойства вызывает изменение визуального представления элемента ввода.

`boolean` **`defaultChecked`**

Для радиокнопок и флажков данное свойство хранит начальное значение атрибута `checked`, элемента. Когда выполняется сброс элементов формы, в свойство `checked` записывается значение этого свойства. Соответствует атрибуту `checked`.

`string` **`defaultValue`**

Для элементов с текстовым значением данное свойство хранит начальное значение, отображаемое элементом. Когда выполняется сброс элементов формы, элемент восстанавливается в это значение. Соответствует атрибуту `value`.

`readonly File[]` **`files`**

Для элементов, значением свойства `type` которых является строка «file», данное свойство ссылается на объект, подобный массиву, хранящий объект или объекты `File`, соответствующие файлам, выбранным пользователем.

string **formAction***

Для кнопок отправки форм это свойство определяет значение, переопределяющее значение свойства `action` вмещающей формы. Соответствует атрибуту `formaction`.

string **formEnctype***

Для кнопок отправки форм это свойство определяет значение, переопределяющее значение свойства `enctype` вмещающей формы. Соответствует атрибуту `formenctype`.

string **formMethod***

Для кнопок отправки форм это свойство определяет значение, переопределяющее значение свойства `method` вмещающей формы. Соответствует атрибуту `formmethod`.

boolean **formNoValidate***

Для кнопок отправки форм это свойство определяет значение, переопределяющее значение свойства `noValidate` вмещающей формы. Соответствует атрибуту `formnovalidate`.

string **formTarget***

Для кнопок отправки форм это свойство определяет значение, переопределяющее значение свойства `target` вмещающей формы. Соответствует атрибуту `formtarget`.

boolean **indeterminate**

Для флажков это свойство определяет, находится ли элемент в неопределенном состоянии (т. е. элемент ни отмечен, ни не отмечен). Это свойство не является отражением какого-либо HTML-атрибута: его можно установить только в сценариях на языке JavaScript.

readonly Element **list***

Элемент `<datalist>`, содержащий элементы `<option>`, которые броузер может использовать в качестве значений для подсказки или автодополнения.

string **max***

Максимальное допустимое значение для данного элемента `Input`.

long **maxLength**

Когда значением свойства `type` является строка «text» или «password», данное свойство определяет максимальное число символов, которые пользователь сможет ввести. Не путайте это свойство со свойством `size`. Соответствует атрибуту `maxlength`.

string **min***

Минимальное допустимое значение для данного элемента `Input`.

boolean **multiple***

Значение `true`, если элемент ввода должен принимать более одного значения указанного типа. Соответствует атрибуту `multiple`.

string **pattern***

Текст регулярного выражения, которому должен соответствовать введенный текст, чтобы его можно было признать допустимым. Это свойство использует синтаксис регулярных выражений JavaScript (без начального и конечного символов слэша), но имейте в виду, что значением этого свойства является строка, а не объект `RegExp`. Отметьте также – чтобы введенный текст был признан допустимым, шаблону должен соответствовать текст целиком, а не только какая-то его часть. (Как если бы шаблон начинался с символа ^ и заканчивался символом $.) Это свойство соответствует атрибуту `pattern`.

string **placeholder**

Короткая текстовая строка, которая должна выводиться в элементе Input, как приглашение к вводу. Когда пользователь передаст элементу фокус ввода, текст приглашения будет стерт и в элементе появится текстовый курсор. Это свойство соответствует атрибуту placeholder.

boolean **readOnly**

Если имеет значение true, элемент будет недоступен для редактирования. Соответствует атрибуту readonly.

boolean **required***

Если имеет значение true, вмещающая форма не будет считаться корректной, пока пользователь не введет значение в данный элемент Input. Соответствует атрибуту required.

readonly Option **selectedOption***

Если свойство list определено и свойство multiple имеет значение false, данное свойство возвращает выбранный элемент Option из списка list, если таковой имеется.

unsigned long **selectionEnd**

Возвращает или изменяет индекс первого символа, следующего за выделенным фрагментом. См. также setSelectionRange().

unsigned long **selectionStart**

Возвращает или изменяет индекс первого символа в выделенном фрагменте в элементе <textarea>. См. также setSelectionRange().

unsigned long **size**

Для текстовых элементов ввода данное свойство определяет ширину элемента в символах. Соответствует атрибуту size. Не путайте со свойством maxLength.

string **step***

Для элементов ввода чисел (включая элементы ввода даты и времени) это свойство определяет шаг изменения значения. Это свойство может быть строкой «any» или вещественным числом. Соответствует атрибуту step.

Date **valueAsDate***

Возвращает значение элемента (см. FormControl) в виде объекта Date.

double **valueAsNumber***

Возвращает значение элемента (см. FormControl) в виде числа.

Методы

В дополнение к методам, перечисленным ниже, элементы Input реализуют также все методы объектов Element и FormControl. Методы, отмеченные звездочкой в этом списке, являются новыми, определяемыми спецификацией HTML5, которые на момент написания этих строк были реализованы не во всех броузерах.

void **select()**

Выделяет весь текст, отображаемый в элементе Input. Во многих броузерах это означает, что при вводе очередного символа выделенный текст будет удален и замещен введенным символом.

void **setSelectionRange**(unsigned long *start*, unsigned long *end*)

Этот метод выделяет текст, отображаемый в элементе Input, начиная с символа в позиции *start* и до (не включая) символа в позиции *end*.

void **stepDown**([long *n*])*

Для элементов, поддерживающих свойство step, уменьшает текущее значение на *n* шагов.

void **stepUp**([long *n*])*

Для элементов, поддерживающих свойство step, увеличивает текущее значение на *n* шагов.

jQuery

jQuery 1.4

библиотека jQuery

Описание

Это краткий справочник по библиотеке jQuery. Более полное описание библиотеки и примеры ее использования приводятся в главе 19. Эта справочная статья организована несколько иначе, чем все остальные. При оформлении сигнатур методов здесь использовались следующие соглашения. Аргументы с именем *sel* являются селекторами библиотеки jQuery. Аргументы с именем *idx* являются целочисленными индексами. Аргументы с именем *elt* или *elts* являются элементами документа или объектами, подобными массивам, содержащими элементы документа. Аргументы с именем *f* являются функциями обратного вызова, а вложенные круглые скобки используются, чтобы показать аргументы, которые передаются библиотекой jQuery при вызове этой функции. Квадратные скобки указывают на необязательные аргументы. Если за именем необязательного аргумента следует знак «равно» и значение, это означает, что указанное значение используется по умолчанию, если аргумент был опущен. Вслед за закрывающей круглой скобкой и двоеточием следует значение, возвращаемое функцией или методом. Если возвращаемое значение не указано, это означает, что метод возвращает объект jQuery, относительно которого он был вызван.

Фабричная функция jQuery

Функция jQuery не только играет роль пространства имен для различных вспомогательных функций, но и является фабричной функцией для создания объектов jQuery. Функция jQuery() может вызываться во всех случаях, перечисленных ниже, но она всегда возвращает объект jQuery, представляющий коллекцию элементов документа (или сам объект Document). Имя $ является псевдонимом для имени jQuery, поэтому во всех формах вызова, представленных ниже, вместо jQuery() можно использовать $():

jQuery(*sel* [, *context*=document])

Возвращает новый объект jQuery, представляющий элементы документа, которые являются потомками для элемента *context* и соответствуют строке селектора *sel*.

jQuery(*elts*)

Возвращает новый объект jQuery, представляющий указанные элементы. Аргумент *elts* может быть единственным элементом документа, массивом или объектом, подобным массиву (таким как NodeList или другой объект jQuery), содержащим элементы документа.

jQuery(*html*, [*props*])

Выполняет синтаксический анализ строки *html* с разметкой HTML и возвращает новый объект jQuery, с одним или более элементами верхнего уровня, содержащимися в строке. Если аргумент *html* содержит единственный HTML-тег, в аргументе *props* можно передать объект, определяющий HTML-атрибуты и обработчики событий для вновь созданного элемента.

`jQuery(f)`

Регистрирует *f* как функцию, которая должна быть вызвана после того, как документ будет загружен и станет доступен для выполнения операций с ним. Если документ уже готов к выполнению операций, функция *f* будет вызвана немедленно, как метод объекта `document`. Возвращает объект `jQuery`, содержащий только объект `document`.

Грамматика селекторов jQuery

Грамматика селекторов jQuery очень похожа на грамматику селекторов CSS3 и подробно описывается в разделе 19.8.1. Далее следует описание грамматики в кратком изложении:

Простые селекторы по имени тега, класса и значению атрибута id

`*` `tagname` `.classname` `#id`

Комбинированные селекторы

`A B`	*B - потомок A*
`A > B`	*B - дочерний по отношению к A*
`A + B`	*B - смежный, следующий за A*
`A ~ B`	*B - смежный по отношению к A*

Фильтры атрибутов

`[attr]`	*имеет атрибут*
`[attr=val]`	*имеет атрибут со значением val*
`[attr!=val]`	*не имеет атрибута со значением val*
`[attr^=val]`	*значение атрибута начинается с val*
`[attr$=val]`	*значение атрибута заканчивается на val*
`[attr*=val]`	*значение атрибута включает val*
`[attr~=val]`	*значение атрибута включает val как слово*
`[attr|=val]`	*значение атрибута начинается с val и необязательного дефиса*

Фильтры по типам элементов

`:button`
`:checkbox`
`:file`
`:header`
`:image`
`:input`
`:password`
`:radio`
`:reset`
`:submit`
`:text`

Фильтры по состоянию элементов

`:animated`
`:checked`
`:disabled`
`:enabled`
`:hidden`
`:selected`
`:visible`

Фильтры по позиции

`:eq(n)`
`:even`
`:first`
`:gt(n)`
`:last`
`:lt(n)`
`:nth(n)`
`:odd`

Фильтры по позиции в документе

`:first-child`
`:last-child`
`:only-child`
`:nth-child(n)`
`:nth-child(even)`
`:nth-child(odd)`
`:nth-child(xn+y)`

Прочие фильтры

`:contains(text)`
`:empty`
`:has(selector)`
`:not(selector)`
`:parent`

Базовые свойства и методы объекта jQuery

Ниже перечислены базовые свойства и методы объектов `jQuery`. Они не влияют на выбор или на выбранные элементы, но позволяют обращаться к выбранным элементам и выполнять итерации по ним. Дополнительные сведения приводятся в разделе 19.1.2.

`context`

Контекст, или корневой элемент, в котором будет производиться выбор. Это второй аргумент функции `$()` или объект `Document`.

`each(`*`f(idx,elt)`*`)`

Вызывает *f* как метод для каждого выбранного элемента. Останавливает итерации, как только функция вернет `false`. Возвращает объект `jQuery`, относительно которого был вызван данный метод.

`get(`*`idx`*`):elt`

`get():array`

Возвращает выбранный элемент с указанным индексом в объекте `jQuery`. Можно также использовать обычный синтаксис индексирования массивов с квадратными скобками. При вызове без аргументов `get()` действует так же, как `toArray()`.

`index():int`

`index(`*`sel`*`):int`

`index(`*`elt`*`):int`

При вызове без аргументов возвращает индекс первого выбранного элемента среди смежных с ним элементов. При вызове с селектором возвращает первый элемент из множества выбранных элементов, соответствующий селектору *sel*, или -1, если такой элемент отсутствует. При вызове с элементом возвращает индекс элемента *elt* в множестве выбранных элементов или -1, если указанный элемент не входит в множество выбранных элементов.

`is(`*`sel`*`):boolean`

Возвращает `true`, если селектору *sel* соответствует хотя бы один выбранный элемент.

`length`

Количество выбранных элементов.

`map(`*`f(idx,elt)`*`):jQuery`

Вызывает *f* как метод для каждого выбранного элемента и возвращает новый объект `jQuery`, хранящий возвращаемые значения, при этом возвращаемые значения `null` и `undefined` не помещаются в массив значений.

`selector`

Оригинальная строка селектора, переданная функции `$()`.

`size():int`

Возвращает значение свойства `length`.

`toArray():array`

Возвращает истинный массив выбранных элементов.

Методы выбора jQuery

Методы, описываемые в этом разделе, изменяют множество выбранных элементов, выполняя фильтрацию, добавляя новые элементы или используя выбранные элементы как начальные точки для нового выбора. В jQuery версии 1.4 и выше выбранные элементы в объекте `jQuery` всегда отсортированы в порядке их следования в документе, а сами множества не содержат дубликатов. Дополнительные сведения приводятся в разделе 19.8.2.

`add(`*`sel`*`, [`*`context`*`])`
`add(`*`elts`*`)`
`add(`*`html`*`)`

Аргументы метода `add()` передаются функции `$()`, а результаты выбора добавляются в текущее множество выбранных элементов.

`andSelf()`

Добавляет в текущий выбор множество ранее выбранных элементов (со стека).

`children([`*`sel`*`])`

Выбирает элементы, являющиеся дочерними, по отношению к выбранным элементам. При вызове без аргументов выбирает все дочерние элементы. При вызове с селектором выбирает только соответствующие ему дочерние элементы.

`closest(`*`sel`*`, [`*`context`*`])`

Выбирает ближайшего предка каждого выбранного элемента, соответствующего селектору *sel*, являющегося потомком по отношению к элементу *context*. Если аргумент *context* опущен, используется свойство `context` объекта `jQuery`.

`contents()`

Выбирает все дочерние элементы во всех выбранных элементах, включая текстовые узлы и комментарии.

`end()`

Выталкивает с внутреннего стека и восстанавливает состояние выбора, предшествовавшее последнему вызову метода, изменяющего выбор.

`eq(`*`idx`*`)`

Выбирает только выбранный элемент с указанным индексом. В jQuery версии 1.4 отрицательные значения отсчитываются с конца множества.

`filter(`*`sel`*`)`
`filter(`*`elts`*`)`
`filter(`*`f`*`(`*`idx`*`):boolean)`

Фильтрует множество выбранных элементов так, что в результат включаются только элементы, которые соответствуют селектору *sel*, или содержатся в объекте *elts*, подобном массиву, или для которых функция-предикат *f* вернет `true`, когда она будет вызвана как метод элемента.

`find(`*`sel`*`)`

Выбирает всех потомков выбранных элементов, которые соответствуют селектору *sel*.

`first()`

Выбирает только первый выбранный элемент.

`has(`*`sel`*`)`
`has(`*`elt`*`)`

Фильтрует выбранные элементы, включая в результат только элементы, которые имеют потомков, соответствующих селектору *sel* или являющихся предками элемента *elt*.

`last()`

Выбирает только последний выбранный элемент.

`next([`*sel*`])`

Выбирает следующий смежный элемент для каждого выбранного элемента. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`nextAll([`*sel*`])`

Выбирает все смежные элементы, следующие за каждым выбранным элементом. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`nextUntil(`*sel*`)`

Выбирает смежные элементы, следующие за каждым выбранным элементом, до (но не включая его) первого смежного элемента, соответствующего селектору *sel*.

`not(`*sel*`)`
`not(`*elts*`)`
`not(`*f*`(`*idx*`):boolean)`

Противоположный методу `filter()`. Фильтрует выбранное множество, исключая элементы, которые соответствуют селектору *sel*, или включены в состав *elts*, или для которых *f* вернет `true`. Аргумент *elts* может быть единственным элементом или объектом, подобным массиву, содержащим элементы. Функция *f* вызывается как метод для каждого выбранного элемента.

`offsetParent()`

Выбирает ближайшего позиционируемого предка для каждого выбранного элемента.

`parent([`*sel*`])`

Выбирает родителя для каждого выбранного элемента. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`parents([`*sel*`])`

Выбирает предков для каждого выбранного элемента. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`parentsUntil(`*sel*`)`

Выбирает предков для каждого выбранного элемента до (но не включая) первого предка, соответствующего селектору.

`prev([`*sel*`])`

Выбирает предшествующий смежный элемент для каждого выбранного элемента. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`prevAll([`*sel*`])`

Выбирает все смежные элементы, предшествующие каждому выбранному элементу. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору.

`prevUntil(`*sel*`)`

Выбирает смежные элементы, предшествующие каждому выбранному элементу, до (но не включая его) первого смежного элемента, соответствующего селектору *sel*.

`pushStack(`*elts*`)`

Помещает текущее состояние выбора на стек, после чего оно может быть восстановлено вызовом метода `end()`, и затем выбирает элементы в массиве (или в объекте, подобном массиву) *elts*.

`siblings([sel])`

Выбирает смежные элементы для каждого выбранного элемента, исключая сам элемент. Если указан аргумент *sel*, исключает из результата элементы, не соответствующие селектору *sel*.

`slice(startidx, [endidx])`

Фильтрует выбор, оставляя только элементы, индексы которых больше или равны *startidx* и меньше (но не равны) *endidx*. Отрицательные индексы отсчитываются от конца множества выбранных элементов. Если аргумент *endidx* не указан, используется значение свойства `length`.

Методы jQuery для работы с элементами

Методы, описываемые здесь, предназначены для чтения и записи значений HTML-атрибутов и свойств CSS-стилей элементов. Функции обратного вызова в аргументе *current* передается текущее значение атрибута или свойства, для которого вычисляется новое значение. Дополнительные сведения приводятся в разделе 19.2.

`addClass(names)`

`addClass(f(idx,current):names)`

Добавляет указанное имя или имена CSS-классов в атрибут `class` каждого выбранного элемента. Или вызывает *f* как метод для каждого элемента для вычисления имени или имен классов, добавляемых в атрибут.

`attr(name):value`

`attr(name, value)`

`attr(name, f(idx,current):value)`

`attr(obj)`

При вызове с единственным строковым аргументом возвращает значение указанного атрибута первого выбранного элемента. При вызове с двумя аргументами устанавливает указанный атрибут во всех выбранных элементах в значение *value* или вызывает *f* как метод для каждого элемента для вычисления значения атрибута. При вызове с единственным аргументом-объектом использует имена его свойств, как имена атрибутов, а значения свойств – как значения атрибутов или как функции вычисления значений атрибутов.

`css(name):value`

`css(name, value)`

`css(name, f(idx,current):value)`

`css(obj)`

Действует подобно методу `attr()`, но возвращает или устанавливает не HTML-атрибуты, а атрибуты CSS-стиля.

`data():obj`

`data(key):value`

`data(key, value)`

`data(obj)`

При вызове без аргументов возвращает объект с данными для первого выбранного элемента. При вызове с одним строковым аргументом возвращает значение указанного свойства объекта с данными. При вызове с двумя аргументами устанавливает указанное свойство в объекте данных для всех выбранных элементов в значение *value*. При вызове с одним аргументом-объектом замещает им объекты с данными во всех выбранных элементах.

hasClass(*name*):boolean

Возвращает *true*, если какой-либо из выбранных элементов содержит имя класса *name* в своем атрибуте class.

height():int
height(*h*)
height(*f*(*idx*,*current*):int)

Возвращает высоту (не включая отступы, рамку и поля) первого выбранного элемента, или устанавливает высоту всех выбранных элементов равной *h* или значению, вычисленному функцией *f*, которая вызывается как метод для каждого элемента.

innerHeight():int

Возвращает высоту плюс отступы для первого выбранного элемента.

innerWidth():int

Возвращает ширину плюс отступы для первого выбранного элемента.

offset():coords
offset(*coords*)
offset(*f*(*idx*,*current*):coords)

Возвращает координаты X и Y (относительно начала документа) первого выбранного элемента или перемещает все выбранные элементы в позицию с координатами *coords* или в позицию, вычисляемую функцией *f*, которая вызывается как метод для каждого выбранного элемента. Координаты определяются в виде объекта со свойствами top и left.

offsetParent():jQuery

Выбирает ближайшего позиционируемого предка для каждого выбранного элемента и возвращает результат в виде нового объекта jQuery.

outerHeight([*margins*=false]):int

Возвращает высоту плюс отступы и рамку, а также поля, если аргумент *margins* имеет значение true, первого выбранного элемента.

outerWidth([*margins*=false]):int

Возвращает ширину плюс отступы и рамку, а также поля, если аргумент *margins* имеет значение true, первого выбранного элемента.

position():coords

Возвращает позицию первого выбранного элемента относительно ближайшего позиционируемого предка. Возвращает объект со свойствами top и left.

removeAttr(*name*)

Удаляет указанный атрибут из всех выбранных элементов.

removeClass(*names*)
removeClass(*f*(*idx*,*current*):names)

Удаляет указанное имя или имена классов из атрибута class всех выбранных элементов. Если в аргументе вместо строки передана функция, она будет вызвана как метод для каждого выбранного элемента, чтобы вычислить имя или имена удаляемых классов.

removeData([*key*])

Удаляет указанное свойство из объекта с данными в каждом выбранном элементе. Если имя свойства не указано, удаляется весь объект с данными целиком.

`scrollLeft():int`
scrollLeft(*int*)

Возвращает позицию горизонтальной полосы прокрутки для первого выбранного элемента или устанавливает ее для всех выбранных элементов.

`scrollTop():int`
scrollTop(*int*)

Возвращает позицию вертикальной полосы прокрутки для первого выбранного элемента или устанавливает ее для всех выбранных элементов.

toggleClass(*names*, [*add*])
toggleClass(*f*(*idx*,*current*):names, [*add*])

Переключает класс с указанным именем или именами в свойстве `class` каждого выбранного элемента. Если указана функция *f*, она будет вызвана как метод для каждого выбранного элемента для вычисления имени или имен переключаемых классов. Если аргумент *add* имеет значение `true` или `false`, добавит или удалит имена классов вместо их переключения.

`val():value`
val(*value*)
val(*f*(*idx*,*current*)):value

Возвращает значение или состояние выбора первого выбранного элемента формы либо устанавливает значение или состояние выбора всех выбранных элементов равным значению аргумента *value* или значению, вычисленному функцией *f*, которая вызывается как метод для каждого элемента.

`width():int`
width(*w*)
width(*f*(*idx*,*current*):int)

Возвращает ширину (не включая отступы, рамку и поля) первого выбранного элемента либо устанавливает ширину всех выбранных элементов равной *w* или значению, вычисленному функцией *f*, которая вызывается как метод для каждого элемента.

Методы jQuery вставки и удаления

Методы, описываемые здесь, вставляют, удаляют или замещают содержимое документа. В сигнатурах методов, приведенных ниже, аргумент *content* может быть объектом `jQuery`, строкой с разметкой HTML или отдельным элементом документа, а аргумент *target* может быть объектом `jQuery`, отдельным элементом документа или строкой селектора. Дополнительные сведения приводятся в разделах 19.2.5 и 19.3.

after(*content*)
after(*f*(*idx*):content)

Вставляет содержимое *content* после каждого выбранного элемента или вызывает *f* как метод и вставляет возвращаемое значение после каждого выбранного элемента.

append(*content*)
append(*f*(*idx*,*html*):content)

Добавляет содержимое *content* в конец каждого выбранного элемента или вызывает *f* как метод и добавляет возвращаемое значение в конец каждого выбранного элемента.

appendTo(*target*):jQuery

Добавляет выбранные элементы в конец каждого элемента, определяемого аргументом *target*, копируя их, если аргумент *target* определяет более одного элемента.

`before(`*`content`*`)`
`before(`*`f`*`(`*`idx`*`):content)`

Действует подобно методу `after()`**, но вставляет содержимое** *`content`* **перед выбранными элементами, а не после.**

`clone([`*`data`*`=false]):jQuery`

Создает полную копию каждого выбранного элемента и возвращает новый объект `jQuery`**, представляющий множество копий элементов. Если аргумент** *`data`* **имеет значение** `true`**, также копирует данные (включая обработчики событий), связанные с выбранными элементами.**

`detach([`*`sel`*`])`

Действует подобно методу `remove()`**, но не удаляет данные, связанные с отсоединенными элементами.**

`empty()`

Удаляет содержимое выбранных элементов.

`html():string`
`html(`*`htmlText`*`)`
`html(`*`f`*`(`*`idx`*`,`*`current`*`):htmlText)`

При вызове без аргументов возвращает содержимое первого выбранного элемента в виде строки с разметкой HTML. При вызове с одним аргументом устанавливает содержимое всех выбранных элементов равным строке *`htmlText`* **или значению, возвращаемому функцией** *`f`*, **которая вызывается как метод этих элементов.**

`insertAfter(`*`target`*`):jQuery`

Вставляет выбранные элементы после каждого элемента, определяемого аргументом *`target`*, **копируя их, если аргумент** *`target`* **определяет более одного элемента.**

`insertBefore(`*`target`*`):jQuery`

Вставляет выбранные элементы перед каждым элементом, определяемым аргументом *`target`*, **копируя их, если аргумент** *`target`* **определяет более одного элемента.**

`prepend(`*`content`*`)`
`prepend(`*`f`*`(`*`idx`*`,`*`html`*`):content)`

Действует подобно методу `append()`**, но вставляет содержимое** *`content`* **в начало каждого выбранного элемента, а не в конец.**

`prependTo(`*`target`*`):jQuery`

Действует подобно методу `appendTo()`**, но вставляет выбранные элементы в начало элементов, определяемых аргументом** *`target`*, **а не в конец.**

`remove([`*`sel`*`])`

Удаляет все выбранные элементы или все выбранные элементы, соответствующие селектору *`sel`*, **из документа, удаляя также все данные, связанные с ними (включая обработчики событий). Обратите внимание, что удаленные элементы исключаются из состава документа, но по-прежнему остаются членами возвращаемого объекта** `jQuery`**.**

`replaceAll(`*`target`*`)`

Вставляет выбранные элементы в документ так, что они замещают каждый элемент, определяемый аргументом *`target`*, **копируя выбранные элементы, если аргумент** *`target`* **определяет более одного элемента.**

replaceWith(*content*)

replaceWith(*f*(*idx*,*html*):content)

Замещает каждый выбранный элемент содержимым *content* или вызывает функцию *f* как метод для каждого выбранного элемента, передавая ей индекс элемента и текущее содержимое в виде разметки HTML, и замещает данный элемент возвращаемым значением.

text():string

text(*plainText*)

text(*f*(*idx*,*current*):plainText)

При вызове без аргументов возвращает содержимое первого выбранного элемента в виде строки с простым текстом. При вызове с одним аргументом устанавливает содержимое всех выбранных элементов равным строке *plainText* или значению, возвращаемому функцией *f*, которая вызывается как метод этих элементов.

unwrap()

Удаляет родителя каждого выбранного элемента, замещая его выбранным элементом и смежными с ним элементами.

wrap(*wrapper*)

wrap(*f*(*idx*):wrapper)

Обертывает каждый выбранный элемент, копируя обертку, если выбранных элементов более одного. Если методу передана функция, она будет вызвана как метод для каждого выбранного элемента, чтобы вычислить обертку. Аргумент *wrapper* может быть элементом, объектом jQuery, селектором или строкой с разметкой HTML, но он должен определять единственный элемент-обертку.

wrapAll(*wrapper*)

Обертывает все выбранные элементы как группу, вставляя обертку *wrapper* в позицию первого выбранного элемента и затем копируя все выбранные элементы в элемент-обертку *wrapper*.

wrapInner(*wrapper*)

wrapInner(*f*(*idx*):wrapper)

Действует подобно методу wrap(), но обертывает элементом *wrapper* (или возвращаемым значением функции *f*) содержимое каждого выбранного элемента, а не сами элементы.

Методы jQuery для работы с событиями

Методы в этом разделе используются для регистрации обработчиков событий и для возбуждения событий. Дополнительные сведения приводятся в разделе 19.4.

event-type()

event-type(*f*(*event*))

Регистрирует *f* как обработчик события типа *event-type* или генерирует событие *event-type*. Библиотека jQuery определяет следующие методы, которые действуют согласно этому шаблону:

ajaxComplete()
ajaxError()
ajaxSend()
ajaxStart()
ajaxStop()
ajaxSuccess()
blur()
change()
click()
dblclick()
error()
focus()
focusin()
focusout()
keydown()
keypress()
keyup()
load()
mousedown()
mouseenter()
mouseleave()
mousemove()
mouseout()
mouseover()
mouseup()
resize()
scroll()
select()
submit()
unload()

`bind(`*type*`, [`*data*`], `*f*`(`*event*`))`
`bind(`*events*`)`

Регистрирует *f* как обработчик событий типа *type* в каждом выбранном элементе. Если указан аргумент *data*, он будет добавлен в объект события перед вызовом *f*. Аргумент *type* может определять несколько типов событий и может включать пространства имен.

Если методу передан единственный объект, он интерпретирует его как отображение типов событий в функции-обработчики и регистрирует обработчики для всех указанных типов событий в каждом выбранном элементе.

`delegate(`*sel*`, `*type*`, [`*data*`], `*f*`(`*event*`))`

Регистрирует *f* как динамический обработчик события. Функция *f* будет вызываться для обработки событий типа *type*, возникающих в элементах, которые определяются селектором *sel* и всплывших до любого из выбранных элементов. Если указан аргумент *data*, он будет добавлен в объект события перед вызовом *f*.

`die(`*type*`, [`*f*`(`*event*`)])`

Отключает динамические обработчики, зарегистрированные методом `live()`, событий типа *type* из элементов, соответствующих селектору, использовавшемуся для создания текущего множества выбранных элементов. Если указана конкретная функция-обработчик *f*, отключает только ее.

`hover(`*f*`(`*event*`))`
`hover(`*enter*`(`*event*`), `*leave*`(`*event*`))`

Регистрирует обработчики событий «mouseenter» и «mouseleave» во всех выбранных элементах. Если указана только одна функция, она будет использована как обработчик обоих событий.

`live(`*type*`, [`*data*`], `*f*`(`*event*`))`

Регистрирует *f* как динамический обработчик события типа *type*. Если указан аргумент *data*, он будет добавлен в объект события перед вызовом *f*. Этот метод не используется для установки обработчиков в выбранные элементы, но он использует строку селектора и контекст данного объекта `jQuery`. Функция *f* будет вызываться, когда события типа *type* будут всплывать до объекта контекста (обычно объект `document`), если целевые элементы события будут соответствовать селектору. См. также `delegate()`.

`one(`*type*`, [`*data*`], `*f*`(`*event*`))`
`one(`*events*`)`

Действует подобно методу `bind()`, но зарегистрированные обработчики событий автоматически отключаются после однократного вызова.

`ready(`*f*`())`

Регистрирует функцию *f*, которая должна быть вызвана, когда документ будет готов к выполнению операций над ним, или вызывает ее немедленно, если документ уже готов. Этот метод не использует выбранные элементы и является синонимом для `$(`*f*`)`.

`toggle(`*f1*`(`*event*`), `*f2*`(`*event*`),...)`

Регистрирует обработчик события «click» во всех выбранных элементах, который циклически переключается между указанными функциями-обработчиками.

`trigger(`*`type`*`, [`*`params`*`])`
`trigger(`*`event`*`)`

Генерирует событие *type* во всех выбранных элементах, передавая *params* обработчикам событий в виде дополнительных параметров. Аргумент *params* можно опустить или передать в нем единственное значение или массив значений. Если передать методу объект события *event*, его свойство `type` будет определять тип события, а все остальные свойства будут скопированы в объект события, который будет передан обработчикам.

`triggerHandler(`*`type`*`, [`*`params`*`])`

Действует подобно методу `trigger()`, но не позволяет всплывать сгенерированному событию или вызывать действия, предусмотренные броузером по умолчанию.

`unbind([`*`type`*`],[`*`f`*`(`*`event`*`)])`

При вызове без аргументов отключает все обработчики событий, зарегистрированные средствами библиотеки jQuery во всех выбранных элементах. При вызове с одним аргументом отключает все обработчики событий типа *type* во всех выбранных элементах. При вызове с двумя аргументами отключает функцию *f*, зарегистрированную как обработчик событий *type* во всех выбранных элементах. Аргумент *type* может представлять несколько типов событий и может включать пространства имен.

`undelegate()`
`undelegate(`*`sel`*`, `*`type`*`, [`*`f`*`(`*`event`*`)])`

При вызове без аргументов отключает все динамические обработчики событий во всех выбранных элементах. При вызове с двумя аргументами отключает динамические обработчики событий типа *type* в элементах, соответствующих селектору *sel*, которые возникают в выбранных элементах. При вызове с тремя аргументами отключает только обработчик *f*.

Методы jQuery воспроизведения визуальных и анимационных эффектов

Методы, описываемые ниже, воспроизводят визуальные и нестандартные анимационные эффекты. Большинство из них возвращают тот же объект `jQuery`, относительно которого они вызывались. Дополнительные сведения приводятся в разделе 19.5.

Параметры анимационных эффектов

`complete duration easing queue specialEasing step`

`jQuery.fx.off`

Установите это свойство в значение `true`, чтобы запретить все визуальные и анимационные эффекты.

`animate(`*`props`*`, `*`opts`*`)`

Воспроизводит эффект, манипулируя CSS-свойствами, определяемыми объектом *props*, в каждом выбранном элементе, используя параметры, определяемые объектом *opts*. Дополнительные сведения об обоих объектах приводятся в разделе 19.5.2.

`animate(`*`props`*`, [`*`duration`*`], [`*`easing`*`], [`*`f`*`()])`

Воспроизводит эффект, манипулируя CSS-свойствами, определяемыми объектом *props*, в каждом выбранном элементе, используя указанную продолжительность *duration* и функцию перехода *easing*. По завершении вызывает *f* как метод для каждого выбранного элемента.

`clearQueue([`*qname*`="fx"])`

Очищает очередь эффектов по умолчанию или указанную очередь для каждого выбранного элемента.

`delay(`*duration*, `[`*qname*`="fx"])`

Добавляет задержку с указанной продолжительностью *duration* **в очередь эффектов по умолчанию или в указанную очередь.**

`dequeue([`*qname*`="fx"])`

Удаляет и вызывает следующую функцию из очереди эффектов по умолчанию или из указанной очереди. Обычно нет необходимости вручную удалять функции из очереди эффектов.

`fadeIn([`*duration*`=400],[`*f*`()])`
`fadeOut([`*duration*`=400],[`*f*`()])`

Воспроизводит в течение *duration* **миллисекунд эффект проявления или растворения элемента, манипулируя его прозрачностью. По завершении вызывает функцию** *f*, **если указана, как метод для каждого выбранного элемента.**

`fadeTo(`*duration*, *opacity*, `[`*f*`()])`

Изменяет CSS-свойство `opacity` **в выбранных элементах до значения** *opacity* **в течение указанной продолжительности** *duration*. **По завершении вызывает функцию** *f*, **если указана, как метод для каждого выбранного элемента.**

`hide()`
`hide(`*duration*, `[`*f*`()])`

При вызове без аргументов немедленно скрывает выбранные элементы. Иначе воспроизводит эффект, уменьшая размеры и непрозрачность всех выбранных элементов так, что они полностью исчезают через *duration* **миллисекунд. По завершении вызывает функцию** *f*, **если указана, как метод для каждого выбранного элемента.**

`slideDown([`*duration*`=400],[`*f*`()])`
`slideUp([`*duration*`=400],[`*f*`()])`
`slideToggle([`*duration*`=400],[`*f*`()])`

Отображает, скрывает или переключает состояние видимости каждого выбранного элемента, изменяя высоту в течение указанной продолжительности *duration*. **По завершении вызывает функцию** *f*, **если указана, как метод для каждого выбранного элемента.**

`show()`
`show(`*duration*, `[`*f*`()])`

При вызове без аргументов немедленно отображает выбранные элементы. Иначе воспроизводит эффект, увеличивая размеры и непрозрачность всех выбранных элементов так, что они становятся полностью видимыми через *duration* **миллисекунд. По завершении вызывает функцию** *f*, **если указана, как метод для каждого выбранного элемента.**

`stop([`*clear*`=false], [`*jump*`=false])`

Останавливает воспроизведение текущего анимационного эффекта (если таковой имеется) во всех выбранных элементах. Если аргумент *clear* **имеет значение** `true`, **также очищает очередь эффектов для каждого элемента. Если аргумент** *jump* **имеет значение** `true`, **перед остановкой присваивает элементам конечные значения, которые должны быть достигнуты в ходе воспроизведения эффекта.**

`toggle([`*`show`*`])`
`toggle(`*`duration`*`, [`*`f`*`()])`

Если аргумент *show* имеет значение `true`, вызывает метод `show()` для немедленного отображения выбранных элементов. Если аргумент *show* имеет значение `false`, вызывает метод `hide()`для немедленного скрытия выбранных элементов. Если аргумент *show* опущен, переключает состояние видимости элементов.

Если указан аргумент *duration*, переключает состояние видимости выбранных элементов, манипулируя размером и прозрачностью в течение *duration* миллисекунд. По завершении вызывает функцию *f*, если указана, как метод для каждого выбранного элемента.

`queue([`*`qname`*`="fx"]):array`
`queue([`*`qname`*`="fx"], `*`f`*`(`*`next`*`))`
`queue([`*`qname`*`="fx"], `*`newq`*`)`

При вызове без аргументов или только с именем очереди возвращает указанную очередь для первого выбранного элемента. При вызове с аргументом-функцией добавляет *f* в указанную очередь для всех выбранных элементов. При вызове с аргументом-массивом замещает указанную очередь для всех выбранных элементов массивом функций *newq*.

Функции jQuery поддержки архитектуры Ajax

Большая часть функциональности библиотеки jQuery, имеющей отношение к поддержке архитектуры Ajax, реализована в виде вспомогательных функций, а не методов. Они являются одними из наиболее сложных функций в библиотеке jQuery. Дополнительные сведения приводятся в разделе 19.6.

Коды состояния Ajax

`success error notmodified timeout parsererror`

Типы данных Ajax

`text html xml script json jsonp`

События Ajax

`ajaxStart ajaxSend ajaxSuccess ajaxError ajaxComplete ajaxStop`

Параметры Ajax

`async`
`beforeSend`
`cache`
`complete`
`contentType`
`context`
`data`
`dataFilter`
`dataType`
`error`
`global`
`ifModified`
`jsonp`
`jsonpCallback`
`password`
`processData`
`scriptCharset`
`success`
`timeout`
`traditional`
`type`
`url`
`username`
`xhr`

`jQuery.ajax(`*`options`*`):XMLHttpRequest`

Сложная, но самая универсальная функция поддержки архитектуры Ajax, на которой основаны все Ajax-утилиты в библиотеке jQuery. Она принимает единственный объект в виде аргумента, свойства которого определяют все тонкости, касающиеся отправки запроса и обработки ответа сервера. Наиболее типичные параметры описываются в разделе 19.6.3.1, а параметры функций обратного вызова – в 19.6.3.2.

`jQuery.ajaxSetup(`*`options`*`)`

Устанавливает указанные параметры как значения по умолчанию. Принимает тот же объект *options*, какой передается функции `jQuery.ajax()`. Указанные вами значения будут использоваться всеми последующими запросами, при оформлении которых не будут явно указаны другие значения параметров. Эта функция не имеет возвращаемого значения.

`jQuery.getJSON(`*url*`, [`*data*`], [`*f*`(`*object*`,`*status*`)]):XMLHttpRequest`

Отправляет асинхронный запрос по адресу *url*, добавляя любые данные *data*. Выполняет синтаксический анализ полученного ответа как строки в формате JSON и передает получившийся объект функции обратного вызова *f*. Возвращает объект `XMLHttpRequest`, если таковой имеется, использовавшийся для выполнения запроса.

`jQuery.getScript(`*url*`, [`*f*`(`*text*`,`*status*`)]):XMLHttpRequest`

Отправляет асинхронный запрос по адресу *url*. При получении ответа выполняет его как сценарий, а затем передает текст ответа функции *f*. Возвращает объект `XMLHttpRequest`, если таковой имеется, использовавшийся для выполнения запроса. Позволяет выполнять междоменные запросы, но в этом случае не передает текст сценария функции *f* и не возвращает объект `XMLHttpRequest`.

`jQuery.get(`*url*`, [`*data*`], [`*f*`(`*data*`,`*status*`,`*xhr*`)], [`*type*`]):XMLHttpRequest`

Отправляет асинхронный HTTP GET-запрос по адресу *url*, добавляя данные *data*, если указаны, в строку параметров запроса данного URL-адреса. При получении ответа интерпретирует его как данные типа *type* или в соответствии со значением заголовка `Content-Type` ответа и выполняет его или выполняет синтаксический анализ, если это необходимо. В заключение передает (возможно, в разобранном виде) данные ответа функции обратного вызова *f* вместе с кодом состояния и объектом `XMLHttpRequest`, использовавшимся для выполнения запроса. Этот объект `XMLHttpRequest`, если имеется, также является возвращаемым значением функции `jQuery.get()`.

`jQuery.post(`*url*`, [`*data*`], [`*f*`(`*data*`,`*status*`,`*xhr*`)], [`*type*`]):XMLHttpRequest`

Действует подобно функции `jQuery.get()`, но выполняет не GET-запрос, а HTTP POST-запрос.

`jQuery.param(`*o*`, [`*old*`=false]):string`

Сериализует имена и значения свойств объекта *o* в формат «www-form-urlencoded», пригодный для добавления в URL-адрес или для передачи в теле HTTP POST-запроса. Большинство функций поддержки Ajax в библиотеке jQuery делают это автоматически, если получают объект в параметре *data*. Если требуется выполнить поверхностную сериализацию объекта в стиле версии jQuery 1.3, во втором аргументе следует передать значение `true`.

`jQuery.parseJSON(`*text*`):object`

Выполняет синтаксический разбор текста в формате JSON и возвращает полученный объект. Функции поддержки архитектуры Ajax в библиотеке jQuery используют эту функцию при запросе данных в формате JSON.

`load(`*url*`, [`*data*`], [`*f*`(`*text*`,`*status*`,`*xhr*`)])`

Отправляет асинхронный запрос по адресу *url*, добавляя любые данные *data*. При получении ответа интерпретирует его как строку с разметкой HTML и вставляет ее в каждый выбранный элемент, замещая любое имеющееся содержимое. В заключение вызывает *f* как метод для каждого выбранного элемента, передавая функции *f* текст ответа, код состояния и объект `XMLHttpRequest`, использовавшийся для выполнения запроса.

Если значение *url* включает пробел, любой текст после пробела используется как селектор и в выбранные элементы вставляется только часть документа в ответе, которая соответствует селектору.

В отличие от большинства функций поддержки архитектуры Ajax в библиотеке jQuery, `load()` является методом, а не функцией. Подобно большинству методов объекта `jQuery`, возвращает объект `jQuery`, относительно которого этот метод был вызван.

`serialize():string`

Сериализует имена и значения выбранных форм или элементов форм и возвращает строку в формате «www-form-urlencoded».

Вспомогательные функции в библиотеке jQuery

Ниже перечислены различные функции и свойства (не методы), имеющиеся в библиотеке jQuery. Дополнительные сведения приводятся в разделе 19.7.

`jQuery.boxModel`

Устаревший синоним для `jQuery.support.boxModel`.

`jQuery.browser`

Это свойство ссылается на объект, идентифицирующий производителя и версию броузера. Объект имеет свойство `msie` в броузере Internet Explorer, `mozilla` – в Firefox, `webkit` – в Safari и Chrome, и `opera` – в Opera. Свойство `version` содержит номер версии броузера.

`jQuery.contains(`*a*`,`*b*`):boolean`

Возвращает `true`, если элемент *a* содержит элемент *b*.

`jQuery.data(`*elt*`):data`

`jQuery.data(`*elt*`, `*key*`):value`

`jQuery.data(`*elt*`, `*data*`)`

`jQuery.data(`*elt*`, `*key*`, `*value*`)`

Низкоуровневая версия метода `data()`. При вызове с единственным элементом *elt* возвращает объект с данными для этого элемента. При вызове с элементом *elt* и строкой *key* возвращает значение с указанным именем из объекта с данными для этого элемента. При вызове с элементом *elt* и объектом *data* устанавливает указанный объект как объект с данными для элемента *elt*. При вызове с элементом *elt*, строкой *key* и значением *value* устанавливает значение с указанным именем в объекте с данными для указанного элемента.

`jQuery.dequeue(`*elt*`, [`*qname*`="fx"])`

Удаляет и вызывает первую функцию из указанной очереди. То же, что и `$(elt).dequeue(qname)`.

`jQuery.each(`*o*`, `*f*`(`*name*`,`*value*`)):o`

`jQuery.each(`*a*`, `*f*`(`*index*`,`*value*`)):a`

Вызывает функцию *f* для каждого свойства объекта *o*, передавая ей имя *name* и значение *value* свойства, при этом функция *f* вызывается как метод значения *value*. Если первый аргумент является массивом или объектом, подобным массиву, вызывает *f* как метод для каждого элемента массива, передавая ей в виде аргументов индекс *index* в массиве и значение *value* элемента. Итерации останавливаются, как только *f* вернет `false`. Эта функция возвращает первый аргумент.

`jQuery.error(`*msg*`)`

Возбуждает исключение с сообщением *msg*. Эту функцию можно вызывать из расширений или переопределить ее для нужд отладки (например, `jQuery.error = alert`).

`jQuery.extend(`*obj*`):object`

`jQuery.extend([`*deep*`=false], `*target*`, `*obj*`...):object`

При вызове с одним аргументом копирует свойства объекта *obj* в глобальное пространство имен библиотеки jQuery. При вызове с двумя и более аргументами копи-

рует свойства второго и всех последующих объектов, в указанном порядке, в объект `target`**. Если необязательный аргумент** `deep` **имеет значение** `true`**, выполняется глубокое копирование и свойства копируются рекурсивно. Возвращает объект, который был дополнен новыми свойствами.**

`jQuery.globalEval(code):void`

Выполняет программный код `code` **на языке JavaScript как сценарий верхнего уровня в теге** `<script>`**. Ничего не возвращает.**

`jQuery.grep(a, f(elt,idx):boolean, [invert=false]):array`

Возвращает новый массив, содержащий только элементы, для которых `f` **вернет** `true`**. Или, если аргумент** `invert` **имеет значение** `true`**, возвращает только элементы, для которых** `f` **вернет** `false`**.**

`jQuery.inArray(v, a):integer`

Выполняет поиск элемента `v` **в массиве** `a` **или в объекте, подобном массиву, и возвращает индекс найденного элемента или -1.**

`jQuery.isArray(x):boolean`

Возвращает `true`**, только если** `x` **является истинным массивом.**

`jQuery.isEmptyObject(x):boolean`

Возвращает `true`**, только если** `x` **не содержит перечислимых свойств.**

`jQuery.isFunction(x):boolean`

Возвращает `true`**, только если** `x` **является функцией.**

`jQuery.isPlainObject(x):boolean`

Возвращает `true`**, только если** `x` **является простым объектом, например, созданным с помощью литерала объекта.**

`jQuery.isXMLDoc(x):true`

Возвращает `true`**, только если** `x` **является XML-документом или элементом XML-документа.**

`jQuery.makeArray(a):array`

Возвращает новый массив, содержащий те же элементы, что и объект `a`**, подобный массиву.**

`jQuery.map(a, f(elt, idx)):array`

Возвращает новый массив, содержащий значения, возвращаемые функцией `f` **для каждого элемента в массиве** `a` **(или в объекте, подобном массиву). Значения** `null`**, возвращаемые функцией** `f`**, игнорируются и не включаются в результирующий массив.**

`jQuery.merge(a,b):array`

Добавляет элементы массива `b` **в конец массива** `a` **и возвращает** `a`**. Аргументы могут быть объектами, подобными массивам, или истинными массивами.**

`jQuery.noConflict([radical=false])`

Восстанавливает значение символа $ в значение, которое он имел перед загрузкой библиотеки jQuery, и возвращает `jQuery`**. Если аргумент** `radical` **имеет значение** `true`**, также восстанавливает значение символа** `jQuery`**.**

`jQuery.proxy(f, o):function`

`jQuery.proxy(o, name):function`

Возвращает функцию, которая вызывает `f` **как метод объекта** `o`**, или функцию, которая вызывает** `o[name]` **как метод объекта** `o`**.**

`jQuery.queue(`*`elt`*`, [`*`qname`*`="fx"], [`*`f`*`])`

Возвращает или создает очередь с указанным именем в элементе *elt* или добавляет новую функцию *f* в эту очередь. То же, что и `$(elt).queue(`*`qname`*`, `*`f`*`)`.

`jQuery.removeData(`*`elt`*`, [`*`name`*`]):void`

Удаляет указанное свойство из объекта с данными в элементе *elt* или удаляет сам объект с данными.

`jQuery.support`

Объект, содержащий множество свойств, описывающих особенности и ошибки, имеющиеся в текущем броузере. В основном представляет интерес для создателей расширений. В броузерах IE свойство `jQuery.support.boxModel` имеет значение `false` при выполнении сценария в режиме совместимости.

`jQuery.trim(`*`s`*`):string`

Возвращает копию строки *s*, из которой удалены начальные и завершающие пробельные символы.

KeyEvent

см. Event

Label

тег <label> для элементов форм **Node, Element**

Объект `Label` представляет тег `<label>` в HTML-форме.

Свойства

`readonly Element` **`control`**

Объект `FormControl`, с которым связан данный объект `Label`. Если определено свойство `htmlFor`, данное свойство будет ссылаться на элемент формы, определяемый свойством `htmlFor`. Иначе это свойство будет ссылаться на первый дочерний элемент `FormControl` данного элемента `<label>`.

`readonly Form` **`form`**

Ссылается на элемент `Form`, содержащий эту метку. Или, если установлен HTML-атрибут `form`, на элемент `Form`, атрибут `id` которого имеет указанное в этом свойстве значение.

`string` **`htmlFor`**

Это свойство соответствует HTML-атрибуту `for`. Поскольку `for` в языке JavaScript является зарезервированным словом, к имени этого свойства добавлен префикс «html», чтобы получился допустимый идентификатор. Если установлено, это свойство должно определять значение атрибута `id` элемента `FormControl`, с которым связана данная метка. (Однако обычно проще вложить элемент `FormControl` в элемент `Label`.)

Link

гиперссылка в HTML-документе **Node, Element**

Ссылки в HTML-документе создаются элементами `<a>`, `<area>` и `<link>`. Теги `<a>` используются в теле документа для создания гиперссылок. Теги `<area>` – это редко используемая возможность, позволяющая создавать «карты изображений». Теги `<link>` используются в разделе `<head>` документа для указания адреса внешних ресурсов, таких

как таблицы стилей и ярлыки. Элементы `<a>` и `<area>` в сценариях на языке JavaScript имеют одинаковое представление. Элементы `<link>` имеют несколько иное представление, но для удобства эти два типа ссылок описываются в одной справочной статье.

Когда объект `Link`, представляющий элемент `<a>`, используется в строковом контексте, он возвращает значение своего свойства `href`.

Свойства

В дополнение к свойствам, перечисленным ниже, объект `Link` также имеет свойства, соответствующие HTML-атрибутам: `hreflang`, `media`, `ping`, `rel`, `sizes`, `target` и `type`. Обратите внимание, что свойства, соответствующие отдельным компонентам URL-адреса (такие как `host` и `pathname`) и возвращающие фрагменты значения свойства `href` ссылки, определены только для элементов `<a>` и `<area>` и отсутствуют в элементах `<link>`, и что свойства `sheet`, `disabled` и `relList` определены только в элементах `<link>`, ссылающихся на таблицы стилей.

`boolean` **`disabled`**

Для элементов `<link>`, ссылающихся на таблицы стилей, определяет, должна ли данная таблица стилей применяться к документу.

`string` **`hash`**

Определяет идентификатор фрагмента документа в значении свойства `href`, включая начальный символ решетки (#), например: «#results».

`string` **`host`**

Определяет имя хоста и порт в значении свойства `href`, например: «*http://www.oreilly.com:1234*».

`string` **`hostname`**

Определяет имя хоста в значении свойства `href`, например: «*http://www.oreilly.com*».

`string` **`href`**

Определяет значение атрибута `href` ссылки. Когда элемент `<a>` или `<area>` используется в строковом контексте, возвращается значение этого свойства.

`string` **`pathname`**

Определяет путь к документу в значении свойства `href`, например: «*/catalog/search.html*».

`string` **`port`**

Определяет порт в значении свойства `href`, например: «1234».

`string` **`protocol`**

Определяет имя протокола в значении свойства `href`, включая завершающее двоеточие, например: «http:».

`readonly DOMTokenList` **`relList`**

Подобно свойству `classList` объекта `Element` это свойство упрощает извлечение, добавление и удаление лексем в HTML-атрибуте `rel` элементов `<link>`.

`string` **`search`**

Определяет строку с параметрами запроса в значении свойства `href`, включая начальный знак вопроса, например: «?q=JavaScript&m=10».

`readonly CSSStyleSheet` **`sheet`**

Для элементов `<link>`, ссылающихся на таблицы стилей, это свойство представляет связанную таблицу стилей.

`string` **`text`**

Простое текстовое содержимое элемента `<a>` или `<area>`. Синоним для свойства `Node.textContent`.

`string` **`title`**

Все HTML-элементы имеют атрибут `title`, который обычно определяет текст всплывающей подсказки для элемента. С помощью этого атрибута или свойства элемента `<link>`, в котором атрибут `rel` имеет значение «alternate stylesheet», можно указать имя таблицы стилей, применение которой пользователь может разрешить или запретить, и если броузер поддерживает альтернативные таблицы стилей, значение свойства `title` может отображаться в интерфейсе броузера в некоторым оформлении.

Location

представляет адрес в броузере и управляет им

Свойство `location` объектов `Window` и `Document` ссылается на объект `Location`, который представляет веб-адрес («местоположение») текущего документа. Свойство `href` содержит полный URL-адрес этого документа, а каждое из оставшихся свойств объекта `Location` описывает фрагмент этого URL-адреса. Эти свойства очень похожи на свойства URL-адреса объекта `Link`. Когда объект `Location` используется в строковом контексте, возвращается значение его свойства `href`. Это означает, что вместо выражения `location.href` можно использовать просто `location`.

Кроме того что объект `Location` представляет текущий URL-адрес, он еще и *управляет* этим адресом. Если строку, содержащую URL-адрес, присвоить объекту `Location` или его свойству `href`, то веб-броузер загрузит документ с указанным URL-адресом и отобразит его. Заставить броузер загрузить новый документ можно также путем изменения части текущего URL-адреса. Например, если установить свойство `search`, броузер перезагрузит текущий URL-адрес с новой строкой запроса. Если установить свойство `hash`, броузер не загрузит новый документ, но создаст новую запись в истории посещений. А если свойство `hash` идентифицирует некоторый фрагмент документа, броузер прокрутит документ так, что указанный элемент окажется в видимой области.

Свойства

Свойства объекта `Location` ссылаются на различные фрагменты URL-адреса текущего документа. Для каждого из следующих свойств дается пример фрагмента следующего (фиктивного) URL-адреса:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

`string` **`hash`**

Содержит якорную часть URL-адреса, включая начальный символ решетки (`#`), в нашем случае – «#results». Эта часть URL-адреса документа определяет имя якорного элемента внутри документа.

`string` **`host`**

Часть URL-адреса, содержащая имя хоста и порт, например: «*http://www.oreilly.com:1234*».

`string` **`hostname`**

Часть URL-адреса, содержащая имя хоста, например: «http://www.oreilly.com».

`string` **`href`**

Полный текст URL-адреса документа, в отличие от других свойств объекта `Location`, которые определяют только части URL-адреса. Присваивание этому свойству нового URL-адреса приводит к тому, что броузер читает и отображает содержимое нового URL-адреса. Непосредственное присваивание объекту `Location` устанавливает это свойство, и при использовании объекта `Location` в строковом контексте возвращается значение этого свойства.

`string` **`pathname`**

Путь в URL-адресе, например: «/catalog/search.html».

`string` **`port`**

Порт в URL-адресе, например: «1234». Обратите внимание, что значением этого свойства является строка, а не число.

`string` **`protocol`**

Протокол в URL-адресе, включая завершающее двоеточие, например: «http:».

`string` **`search`**

Часть URL-адреса, которая содержит строку запроса, включая начальный вопросительный знак, например: «?q=JavaScript&m=10».

Методы

`void` **`assign`**`(string` *`url`*`)`

Загружает и отображает содержимое адреса *`url`*, как если бы значение *`url`* было присвоено свойству `href`.

`void` **`reload`**`()`

Повторно загружает текущий документ.

`void` **`replace`**`(string` *`url`*`)`

Загружает и отображает содержимое адреса *`url`*, замещая текущий документ в истории посещений, вследствие чего щелчок на кнопке Back броузера не вернет его к предыдущему документу.

MediaElement

элемент проигрывателя **Node, Element**

`MediaElement` является общим суперклассом для элементов `<audio>` и `<video>`. Эти два элемента определяют практически идентичные прикладные интерфейсы, описываемые здесь, тем не менее просмотрите справочные статьи Audio и Video, где приводится описание дополнительных особенностей аудио- и видеопроигрывателей. А также обратитесь к разделу 21.2, где дается введение в эти мультимедийные элементы.

Константы

Константы `NETWORK` определяют возможные значения свойства `networkState`, а константы `HAVE` – возможные значения свойства `readyState`.

`unsigned short` **`NETWORK_EMPTY = 0`**

Элемент еще не приступил к использованию сети. Это состояние предшествует установке атрибута `src`.

`unsigned short` **`NETWORK_IDLE = 1`**

В настоящий момент элемент не производит загрузку данных из сети. Возможно, он уже загрузил ресурс полностью или загрузил необходимый объем данных в буфер. Или, возможно, свойство `preload` установлено в значение «none», и пока не была запрошена загрузка или проигрывание данных.

`unsigned short` **`NETWORK_LOADING = 2`**

В настоящее время элемент загружает данные из сети.

`unsigned short` **`NETWORK_NO_SOURCE = 3`**

Элемент не использует сеть, потому что не способен отыскать источник с данными для проигрывания.

`unsigned short` **`HAVE_NOTHING = 0`**

Мультимедийные данные или метаданные еще не были загружены.

`unsigned short` **`HAVE_METADATA = 1`**

Метаданные были загружены, но данные для текущей позиции проигрывания еще не были загружены. Это означает, что можно узнать продолжительность или размеры кадра видеозаписи, а также перейти к другой позиции проигрывания, изменив значение свойства `currentTime`, но броузер в настоящее время не проигрывает данные в позиции `currentTime`.

`unsigned short` **`HAVE_CURRENT_DATA = 2`**

Данные для текущей позиции проигрывания `currentTime` были загружены, но данных пока недостаточно, чтобы можно было начать проигрывание. Для видеозаписей это обычно означает, что текущий кадр уже загружен, а следующий – еще нет. Это состояние возникает в конце аудио- или видеозаписи.

`unsigned short` **`HAVE_FUTURE_DATA = 3`**

Загружен объем данных, достаточный, чтобы начать проигрывание, но, скорее всего, недостаточный, чтобы проиграть запись до конца без приостановки для загрузки дополнительных данных.

`unsigned short` **`HAVE_ENOUGH_DATA = 4`**

Загружен объем данных, достаточный, чтобы броузер смог проиграть запись до конца без приостановки.

Свойства

`boolean` **`autoplay`**

Если имеет значение `true`, проигрыватель автоматически начнет проигрывание, когда будет загружен достаточный объем данных. Соответствует HTML-атрибуту `autoplay`.

`readonly TimeRanges` **`buffered`**

Фрагменты уже загруженных в буфер данных.

`boolean` **`controls`**

Если имеет значение `true`, проигрыватель должен отобразить элементы управления проигрыванием. Соответствует HTML-атрибуту `controls`.

`readonly string` **`currentSrc`**

URL-адрес мультимедийных данных, полученный из атрибута `src` или из одного из дочерних элементов `<source>`, или пустая строка, если данные для проигрывания не указаны.

`double` **`currentTime`**

Текущая позиция проигрывания в секундах. Установка этого свойства позволяет перейти к другой позиции проигрывания.

`double` **`defaultPlaybackRate`**

Скорость проигрывания, используемая при проигрывании в нормальном режиме. Значение по умолчанию 1.0.

`readonly double` **`duration`**

Продолжительность записи в секундах. Если продолжительность неизвестна (например, когда метаданные еще не были загружены), это свойство имеет значение `NaN`. Если проигрываются потоковые данные с неопределенной продолжительностью, это свойство имеет значение `Infinity`.

`readonly boolean` **`ended`**

Имеет значение `true`, если достигнут конец записи.

`readonly MediaError` **`error`**

Это свойство устанавливается, когда возникает ошибка, в противном случае имеет значение `null`. Ссылается на объект, свойство `code` которого описывает тип ошибки.

`readonly double` **`initialTime`**

Начальная позиция проигрывания в секундах. Обычно имеет значение 0, но в некоторых случаях (например, когда проигрываются потоковые данные) может иметь различные значения.

`boolean` **`loop`**

Если имеет значение `true`, проигрыватель должен автоматически перезапускать воспроизведение записи по достижении конца. Это свойство соответствует HTML-атрибуту `loop`.

`boolean` **`muted`**

Определяет, должна ли воспроизводиться запись без звука. Это свойство можно использовать, чтобы отключать и включать звук. Для элементов `<video>` можно использовать атрибут `audio="muted"`, чтобы отключить звук по умолчанию.

`readonly unsigned short` **`networkState`**

Определяет, загружаются данные в настоящий момент или нет. Допустимые значения перечислены в разделе «Константы» выше.

`readonly boolean` **`paused`**

Имеет значение `true`, если в настоящий момент проигрывание приостановлено.

`double` **`playbackRate`**

Текущая скорость проигрывания. 1.0 – нормальная скорость проигрывания. Значения больше 1.0 соответствуют ускоренной скорости проигрывания вперед. Значения от 0 до 1.0 соответствуют замедленной скорости проигрывания вперед. Значения меньше 0 соответствуют проигрыванию в обратном направлении. (Звук всегда отключается при проигрывании в обратном направлении, а также при слишком быстром или слишком медленном проигрывании вперед.)

`readonly TimeRanges` **`played`**

Фрагменты, которые уже были проиграны.

`string` **`preload`**

Это свойство соответствует HTML-атрибуту с тем же именем, и его можно использовать, чтобы указать, какой объем данных должен загрузить броузер, прежде чем пользователь сможет запустить проигрывание. Значение «none» означает, что предварительная загрузка данных не должна выполняться. Значение «metadata» означает, что броузер должен предварительно загрузить метаданные (такие как продолжительность), но не фактические данные. Значение «auto» (или просто пустая строка, если атрибут `preload` указан без значения) означает, что броузер может загрузить весь ресурс целиком на тот случай, если пользователь решит проиграть его.

`readonly unsigned short` **`readyState`**

Определяет готовность данных к проигрыванию, исходя из объема данных, загруженных в буфер. Допустимые значения определяются константами `HAVE_`, описанными выше.

`readonly TimeRanges` **`seekable`**

Фрагмент или фрагменты, значения времени для которых могут быть присвоены свойству `currentTime`. При проигрывании простых файлов обычно можно установить любое значение от 0 до значения свойства `duration`. Но для потоковых данных позиция в прошлом может отсутствовать в буфере, а позиция в будущем может быть еще недоступна.

`readonly boolean` **`seeking`**

Имеет значение `true`, пока элемент проигрывателя выполняет переход к новой позиции проигрывания `currentTime`. Если данные для новой позиции проигрывания уже загружены в буфер, это свойство будет иметь значение `true` очень короткий промежуток времени. Но если для перехода проигрывателю необходимо загрузить новые данные, свойство `seeking` будет оставаться в значении `true` довольно продолжительное время.

`string` **`src`**

Соответствует HTML-атрибуту `src` элемента проигрывателя. Присваивание нового значения этому свойству заставит проигрыватель загрузить новые данные для проигрывания. Не путайте это свойство со свойством `currentSrc`.

`readonly Date` **`startOffsetTime`**

Действительные дата и время позиции проигрывания 0, если метаданные содержат такую информацию. (Видеофайл может содержать время съемки, например.)

`double` **`volume`**

Определяет уровень громкости воспроизводимой аудиозаписи. Значение должно быть в диапазоне от 0 до 1. См. также описание свойства `muted`.

Обработчики событий

Теги `<audio>` и `<video>` определяют следующие обработчики событий, которые можно устанавливать как HTML-атрибуты или как JavaScript-свойства. На момент написания этих строк некоторые броузеры не поддерживали эти свойства и требовали, чтобы обработчики событий регистрировались с помощью метода `addEventListener()` (`EventTarget`). События элементов проигрывателей не всплывают, и для них не предусмотрено действий по умолчанию, которые можно было бы отменить. Связанные с ними объекты событий являются обычными объектами `Event`.

Обработчик событий	Вызывается, когда...
`onabort`	Элемент прекратил загрузку данных, вероятно, по запросу пользователя. Свойство `error.code` имеет значение `error.MEDIA_ERR_ABORTED`.
`oncanplay`	Загружено достаточно данных, чтобы начать проигрывание, но наверняка потребуется загрузка дополнительных данных.
`oncanplaythrough`	Загружено достаточно данных, чтобы проигрывание не приостанавливалось на загрузку дополнительных данных.
`ondurationchange`	Изменилось значение свойства `duration`.

Обработчик событий	Вызывается, когда...
onemptied	Свойство networkState получило значение NETWORK_EMPTY вследствие ошибки или остановки проигрывателя.
onended	Проигрывание остановлено по достижении конца записи.
onerror	Сетевая или какая-то другая ошибка препятствует загрузке данных. Свойство error.code имеет значение, отличное от MEDIA_ERR_ABORTED (MediaError).
onloadeddata	Данные для текущей позиции проигрывания загружены в первый раз.
onloadedmetadata	Были загружены метаданные, и стали доступны продолжительность и размеры кадра.
onloadstart	Элемент послал запрос на загрузку данных.
onpause	Был вызван метод pause(), и проигрывание было приостановлено.
onplay	Был вызван метод play(), или атрибут autoplay вызвал запуск проигрывания.
onplaying	Данные проигрываются.
onprogress	Загрузка данных из сети продолжается. Обычно генерируется от 2 до 8 раз в секунду. Обратите внимание, что объект, связанный с этим событием, является обычным объектом Event, а не ProgressEvent, используемым другими прикладными интерфейсами, которые возбуждают события с именем «progress».
onratechange	Изменилось значение свойства playbackRate или defaultPlaybackRate.
onseeked	Свойство seeking опять получило значение false.
onseeking	Сценарий или пользователь потребовал выполнить переход к позиции проигрывания, для которой данные еще не были загружены, вследствие чего проигрывание было приостановлено до загрузки данных. Свойство seeking имеет значение true.
onstalled	Элемент пытается загрузить данные, но данные не поступают.
onsuspend	Элемент загрузил в буфер достаточно большой объем данных и временно приостановил загрузку.
ontimeupdate	Изменилось значение свойства currentTime. При обычном проигрывании это событие возбуждается от 4 до 60 раз в секунду.
onvolumechange	Изменилось значение свойства volume или muted.
onwaiting	Проигрывание не может быть начато, или проигрывание было приостановлено из-за недостаточного объема буферизованных данных. Когда будет загружен достаточный объем данных, последует событие «playing».

Методы

string **canPlayType**(string *type*)

Этот метод запрашивает у элемента проигрывателя, способен ли он проигрывать данные MIME-типа *type*. Если проигрыватель точно определит, что не может проигрывать данные указанного типа, он вернет пустую строку. Если проигрыватель полагает (но не уверен), что может проигрывать данные указанного типа, он вернет строку «probably» («возможно»). В общем случае элементы проигрывателя никогда не вернут строку «probably», если *type* не включает параметр codecs= со списком мультимедийных кодеков. Если проигрыватель не уверен, что может проигрывать данные указанного типа, этот метод вернет строку «maybe».

void **load()**

Этот метод сбрасывает элемент проигрывателя в исходное состояние и заставляет его выбрать источник данных и начать загрузку. Это происходит автоматически, когда элемент впервые вставляется в документ, и всякий раз, когда изменяется значение атрибута `src`. Однако при добавлении, удалении или изменении вложенных элементов `<source>` метод `load()` необходимо вызывать явно.

void **pause()**

Приостанавливает проигрывание.

void **play()**

Начинает проигрывание записи.

MediaError

представляет ошибку в элементе <audio> или <video>

Когда в элементе `<audio>` или `<video>` возникает ошибка, генерируется событие «error» и в свойстве `error` объекта события обработчику передается объект `MediaError`. Свойство `code` этого объекта определяет тип возникшей ошибки. Возможные значения этого свойства определяют приведенные ниже константы.

Константы

unsigned short **MEDIA_ERR_ABORTED = 1**

Пользователь остановил загрузку данных.

unsigned short **MEDIA_ERR_NETWORK = 2**

Мультимедийные данные имеют корректный тип, но сетевая ошибка препятствует их загрузке.

unsigned short **MEDIA_ERR_DECODE = 3**

Мультимедийные данные имеют корректный тип, но ошибка кодирования препятствует их декодированию и проигрыванию.

unsigned short **MEDIA_ERR_SRC_NOT_SUPPORTED = 4**

Тип мультимедийных данных, на которые ссылается атрибут `src`, не поддерживаются броузером.

Свойства

readonly unsigned short **code**

Это свойство описывает тип возникшей ошибки. Его значением может быть одна из констант, перечисленных выше.

MessageChannel

пара соединенных объектов MessagePorts

Объект `MessageChannel` представляет пару соединенных друг с другом объектов `MessagePort`. Вызов метода `postMessage()` в любом из них сгенерирует событие «message» в другом. Если в программе потребуется создать частный канал связи с окном `Window` или фоновым потоком выполнения `Worker`, можно создать объект `MessageChannel` и затем передать один объект `MessagePort` из этой пары окну или потоку выполнения (используя аргумент *ports* метода `postMessage()`).

Типы `MessageChannel` и `MessagePort` являются нововведением, появившимся в спецификации HTML5, и на момент написания этих строк некоторые броузеры поддерживали междоменный обмен сообщениями (раздел 22.3) и фоновые потоки выполнения (раздел 22.4) без применения частных каналов связи на основе объектов `MessagePort`.

Конструктор

new **MessageChannel()**

Этот конструктор, не имеющий аргументов, возвращает новый объект `MessageChannel`.

Свойства

readonly MessagePort **port1**

readonly MessagePort **port2**

Два соединенных друг с другом порта, образующих канал обмена данными. Оба порта являются совершенно равноценными: достаточно просто один сохранить в своем программном коде, а другой передать окну `Window` или фоновому потоку выполнения `Worker`, с которым требуется организовать обмен данными.

MessageEvent

сообщение из другого контекста выполнения **Event**

Различные прикладные интерфейсы используют события «message» для организации асинхронных взаимодействий между независимыми контекстами выполнения. Все объекты – `Window`, `Worker`, `WebSocket`, `EventSource` и `MessagePort` – определяют свойство `onmessage` для регистрации обработчика события «message». Сообщение, связанное с событием «message», может быть любым значением, допустимым в языке JavaScript, которое можно скопировать, как описывается во врезке «Структурированные копии» в главе 22. Сообщение заключается в объект `MessageEvent` и доступно в виде свойства `data`. Различные прикладные интерфейсы, опирающиеся на событие «message», могут определять дополнительные свойства в объекте `MessageEvent`. События «message» не всплывают, и для них не предусмотрено действий по умолчанию, которые можно было бы отменить.

Свойства

readonly any **data**

Это свойство хранит доставленное сообщение. Свойство `data` может иметь значение любого типа, которое можно скопировать с применением алгоритма структурированного копирования (врезка «Структурированные копии» в главе 22). К ним относятся значения базового JavaScript, включая объекты и массивы, но не функции. Некоторые значения клиентского JavaScript, такие как узлы `Document` и `Element`, не могут передаваться, но могут передаваться объекты `Blob` и `ArrayBuffer`.

readonly string **lastEventId**

Для событий «message» в интерфейсе `EventSource` (раздел 18.3) это поле содержит строку `lastEventId`, если имеется, отправленную сервером.

readonly string **origin**

Для событий «message» в интерфейсах `EventSource` (раздел 18.3) или `Window` (раздел 22.3) это свойство содержит URL-адрес отправителя сообщения.

readonly MessagePort[] **ports**

Для событий «message» в интерфейсах `Window` (раздел 22.3), `Worker` (раздел 22.4) и `MessagePort` это свойство содержит массив объектов `MessagePort`, если он был передан соответствующему вызову `postMessage()`.

readonly Window **source**

Для событий «message» в интерфейсе `Window` (раздел 22.3) это свойство ссылается на объект `Window`, отправивший сообщение.

MessagePort

передает асинхронные сообщения **EventTarget**

Объект `MessagePort` используется для передачи асинхронных сообщений в виде событий, обычно между различными контекстами выполнения, такими как окна или фоновые потоки выполнения. Объекты `MessagePort` должны использоваться в виде связанных пар: см. `MessageChannel`. Вызов метода `postMessage()` объекта `MessagePort` генерирует событие «message» в связанном с ним объекте `MessagePort`. Прикладной интерфейс обмена междоменными сообщениями (раздел 22.3) и фоновые потоки выполнения (раздел 22.4) также взаимодействуют с использованием объектов `postMessage()` и событий `message`. Эти прикладные интерфейсы фактически неявно используют объект `MessagePort`. Явное использование объектов `MessageChannel` и `MessagePort` позволяет создавать дополнительные, частные каналы обмена данными и может применяться, например, для организации непосредственных взаимодействий двух соседних фоновых потоков выполнения.

Типы `MessageChannel` и `MessagePort` являются нововведением, появившимся в спецификации HTML5, и на момент написания этих строк некоторые броузеры поддерживали междоменный обмен сообщениями (раздел 22.3) и фоновые потоки выполнения (раздел 22.4) без применения частных каналов связи на основе объектов `MessagePort`.

Методы

void **close()**

Отключает данный объект `MessagePort` от порта, к которому он был подключен (если таковой имеется). Последующие вызовы метода `postMessage()` не будут иметь никакого эффекта, и в будущем сообщения «message» приходить не будут.

void **postMessage(**any *message*, [MessagePort[] *ports*]**)**

Отправляет копию сообщения *message* через порт и передает его в форме события «message» порту, с которым соединен данный порт. Если указан аргумент *ports*, его значение также будет доставлено вместе с событием «message». Аргумент *message* может иметь любое значение, совместимое с алгоритмом структурированного копирования (врезка «Структурированные копии» в главе 22).

void **start()**

Запускает механизм возбуждения событий «message» в объекте `MessagePort`. До вызова этого метода все данные, отправляемые через порт, будут сохраняться в буфере. Подобная задержка событий позволяет сценариям зарегистрировать все обработчики событий до того, как будет отправлено хоть одно сообщение. Имейте, однако, в виду, что вызывать этот метод необходимо только при использовании метода `addEventListener()` интерфейса `EventTarget`. Если сценарий регистрирует обработчик посредством свойства `onmessage`, метод `start()` будет вызван неявно.

Обработчики событий

`onmessage`

Это свойство определяет обработчик событий «message». События «message» генерируются в объекте `MessagePort`. Они не всплывают, и для них не предусматривается действий по умолчанию. Обратите внимание, что при установке этого свойства вызывается метод `start()`, который запускает механизм возбуждения событий «message».

Meter

графический индикатор, или шкала **Node, Element**

Объект `Meter` представляет HTML-элемент `<meter>`, отображающий графическое представление значения внутри диапазона возможных значений, где диапазон возможных значений может быть произвольно разбит на области низких, средних и высоких значений.

Большинство свойств этого объекта просто являются отражением HTML-атрибутов с теми же именами. Однако свойства объекта являются числовыми, тогда как HTML-атрибуты – строками.

Элемент `<meter>` был введен спецификацией HTML5, поэтому на момент написания этих строк он поддерживался не всеми броузерами.

Свойства

`readonly Form` **`form`**

Элемент `Form`, если имеется, являющийся предком для данного элемента или определяемый HTML-атрибутом `form`.

`double` **`high`**

Если определено, значения в диапазоне между `high` и `max` будут графически отнесены к «высоким».

`readonly NodeList` **`labels`**

Объект, подобный массиву, содержащий элементы `Label`, связанные с этим элементом.

`double` **`low`**

Если определено, значения в диапазоне между `low` и `max` будут графически отнесены к «низким».

`double` **`max`**

Максимальное значение, которое может отображать элемент `<meter>`. По умолчанию имеет значение 1.

`double` **`min`**

Минимальное значение, которое может отображать элемент `<meter>`. По умолчанию имеет значение 0.

`double` **`optimum`**

Определяет значение, которое следует считать оптимальным.

`double` **`value`**

Значение, которое представляет данный элемент `<meter>`.

MouseEvent

см. Event

Navigator

информация о веб-броузере

Объект `Navigator` содержит свойства, описывающие используемый веб-броузер. Посредством этих свойств можно выполнить специфическую для платформы настройку сценария. Имя этого объекта, очевидно, ссылается на броузер Netscape Navigator, тем не менее все броузеры поддерживают этот объект. Имеется только один экземпляр объекта `Navigator`, к которому можно обращаться через свойство `navigator` любого объекта `Window`.

Исторически объект `Navigator` использовался для «опознавания клиента», позволяя выбирать, какой фрагмент программного кода выполнять в зависимости от типа броузера. В примере 14.3 демонстрируется простейший прием, а в сопутствующем ему описании в разделе 14.4 упоминается масса ловушек, которые имеются в объекте `Navigator`. Наилучший способ проверки совместимости, не зависящий от типа броузера, описывается в разделе 13.4.3.

Свойства

`readonly string` **`appName`**

Имя броузера. Для броузеров, созданных на основе Netscape, значение этого свойства равно «Netscape». В Internet Explorer оно равно «Microsoft Internet Explorer». Для совместимости с существующим программным кодом многие броузеры возвращают старую, не соответствующую действительности, информацию.

`readonly string` **`appVersion`**

Информация о версии и платформе броузера. Для совместимости с существующим программным кодом в большинстве броузеров это свойство возвращает устаревшую информацию.

`readonly Geolocation` **`geolocation`**

Ссылка на объект `Geolocation` для данного броузера. Методы этого объекта позволяют сценариям запрашивать текущее географическое местонахождение пользователя.

`readonly boolean` **`onLine`**

Имеет значение `false`, если броузер не будет предпринимать попыток загрузить что-либо из сети. Это может быть обусловлено тем, что броузер выполняется на компьютере, не подключенном к сети, или тем, что пользователь настроил броузер на автономную работу. Если броузер будет предпринимать пытаться загрузить что-либо (потому что компьютер подключен к сети), это свойство будет иметь значение `true`. Когда значение этого свойства изменяется, броузер возбуждает события «online» и «offline» в объекте `Window`.

`readonly string` **`platform`**

Название операционной системы и/или аппаратной платформы, на которой выполняется броузер. Стандартный набор значений для этого свойства не определен, но вот некоторые типичные значения: «Win32», «MacPPC» и «Linux i586».

`readonly string` **`userAgent`**

Значение, которое броузер подставляет в заголовок user-agent в HTTP-запросах. Например:

```
Mozilla/5.0 (X11; U; Linux i686; en-US)
AppleWebKit/534.16 (KHTML, like Gecko)
Chrome/10.0.648.45
Safari/534.16
```

Методы

void **registerContentHandler**(string *mimeType*, string *url*, string *title*)

Этот метод выполняет запрос на регистрацию *url* как обработчика, используемого для отображения содержимого типа *mimeType*. Аргумент *title* – заголовок сайта, который может отображаться перед пользователем. Аргумент *url* должен содержать строку «%s». Когда этот обработчик содержимого должен будет использоваться для обработки веб-страницы указанного типа *mimeType*, URL-адрес этой веб-страницы будет закодирован и вставлен в *url* на место «%s». Затем броузер перейдет по получившемуся URL-адресу. Это новый метод, введенный спецификацией HTML5, и он может быть реализован не во всех броузерах.

void **registerProtocolHandler**(string *scheme*, string *url*, string *title*)

Действует подобно методу registerContentHandler(), но регистрирует веб-сайт для использования в качестве обработчика схемы протокола *scheme*, указанного в URL-адресе. Аргумент *scheme* должен быть строкой, такой как «mailto» или «sms» без двоеточия. Это новый метод, введенный спецификацией HTML5, и он может быть реализован не во всех броузерах.

void **yieldForStorageUpdates**()

Сценарии, использующие объекты Document.cookie, Window.localStorage и Window.sessionStorage (см. Storage и главу 20), не имеют возможности определять, производятся ли изменения в хранилище параллельно выполняющимися в разных окнах сценариями (с тем же происхождением). Броузеры могут (хотя на момент написания этих строк такую возможность поддерживали не все броузеры) предотвратить параллельные изменения с помощью механизма блокировок, подобного тому, что используется в базах данных. В броузерах, поддерживающих такую возможность, этот метод неявно освобождает блокировку, предоставляя возможность сценариям в других окнах сохранить свои изменения. Значения, извлекаемые из хранилища после вызова этого метода, могут отличаться от тех, что извлекались перед его вызовом.

Node

Все объекты в дереве документа (включая сам объект Document) реализуют интерфейс Node, который предоставляет фундаментальные свойства и узлы для выполнения манипуляций с деревом. Свойство parentNode и массив childNodes[] позволяют передвигаться вверх и вниз по дереву документа. Можно перечислить дочерние узлы данного узла, выполнив цикл по элементам childNodes[] или используя свойства firstChild и nextSibling (или свойства lastChild и previousSibling для обхода в обратном порядке). Методы appendChild(), insertBefore(), removeChild() и replaceChild() позволяют модифицировать дерево документа, изменяя дочерние узлы данного узла.

Каждый объект в дереве документа реализует как интерфейс Node, так и более специализированный интерфейс, например, Element или Text. Свойство nodeType указывает, какой подынтерфейс реализует узел. Это свойство позволяет проверить тип узла перед тем, как использовать свойства и методы более специализированного интерфейса. Например:

```
var n;                      // Содержит узел, с которым выполняются операции
if (n.nodeType == 1) {      // Или использовать константу Node.ELEMENT_NODE
```

```
        var tagname = n.tagName; // Если узел является узлом Element, это имя тега
    }
```

Константы

```
unsigned short ELEMENT_NODE = 1
unsigned short TEXT_NODE = 3
unsigned short PROCESSING_INSTRUCTION_NODE = 7
unsigned short COMMENT_NODE = 8
unsigned short DOCUMENT_NODE = 9
unsigned short DOCUMENT_TYPE_NODE = 10
unsigned short DOCUMENT_FRAGMENT_NODE = 11
```

Эти константы определяют возможные значения свойства `nodeType`. Обратите внимание, что они являются статическими свойствами функции-конструктора `Node()` – они не являются свойствами отдельных объектов `Node`. Отметьте также, что они не определены в IE версии 8 и ниже. Для совместимости в сценариях можно использовать числовые значения констант или определить собственные константы.

```
unsigned short DOCUMENT_POSITION_DISCONNECTED = 0x01
unsigned short DOCUMENT_POSITION_PRECEDING = 0x02
unsigned short DOCUMENT_POSITION_FOLLOWING = 0x04
unsigned short DOCUMENT_POSITION_CONTAINS = 0x08
unsigned short DOCUMENT_POSITION_CONTAINED_BY = 0x10
```

Эти константы определяют биты, которые могут быть установлены или сброшены в значении, возвращаемом методом `compareDocumentPosition()`.

Свойства

readonly string **baseURI**

Определяет базовый URL-адрес для данного объекта `Node`, который будет использоваться при разрешении относительных URL-адресов. Для всех узлов в HTML-документах этот URL-адрес определяется элементом `<base>` или свойством `Document.URL`, из значения которого исключается идентификатор фрагмента.

readonly NodeList **childNodes**

Это свойство является объектом, подобным массиву, содержащим дочерние узлы текущего узла. Это свойство никогда не должно иметь значение `null`: для узлов, не имеющих дочерних узлов, `childNodes` – это массив со свойством `length`, равным нулю. Обратите внимание: объект `NodeList` «живой», т. е. любое изменение в списке дочерних узлов элемента немедленно становится видимым через объект `NodeList`.

readonly Node **firstChild**

Первый дочерний узел этого узла или `null`, если узел не имеет дочерних узлов.

readonly Node **lastChild**

Последний дочерний узел этого узла или `null`, если узел не имеет дочерних узлов.

readonly Node **nextSibling**

Смежный узел, непосредственно следующий за данным узлом в массиве `childNodes[]` узла `parentNode`, или `null`, если такого узла нет.

readonly string **nodeName**

Имя узла. Для узлов `Element` определяет имя тега элемента, которое может быть также получено с помощью свойства `tagName` интерфейса `Element`. Для большинства других типов узлов значение является строковой константой, зависящей от типа узла.

readonly unsigned short **nodeType**

Тип узла, т.е. информация о том, какой подынтерфейс реализует узел. Допустимые значения определяются перечисленными выше константами. Однако т.к. эти константы не поддерживаются в Internet Explorer, вместо них могут использоваться числовые значения. В HTML-документах распространенные значения для этого свойства таковы: 1 – для узлов `Element`, 3 – для узлов `Text`, 8 – для узлов `Comment` и 9 – для единственного узла `Document` верхнего уровня.

string **nodeValue**

Значение узла. Для узлов `Text` содержит текстовое содержимое.

readonly Document **ownerDocument**

Объект `Document`, частью которого является данный узел. Для узлов `Document` это свойство равно `null`. Обратите внимание, что узлы всегда имеют владельца, даже если они не были добавлены в документ.

readonly Node **parentNode**

Родительский узел (или узел-контейнер) этого узла или `null`, если родительского узла не существует. Обратите внимание: узлы `Document` и `DocumentFragment` никогда не имеют родительских узлов. Кроме того, в узлах, удаленных из документа, а также в только что созданных, но еще не вставленных в дерево документа узлах свойство `parentNode` равно `null`.

readonly Node **previousSibling**

Смежный узел, непосредственно предшествующий данному узлу в массиве `childNodes[]` родительского узла `parentNode`, или `null`, если такого узла нет.

string **textContent**

Для узлов `Text` и `Comment` это свойство является синонимом свойства `data`. Для узлов `Element` и `DocumentFragment` это свойство возвращает объединенное содержимое всех вложенных узлов `Text`. Операция присваивания этому свойству узлов `Element` или `DocumentFragment` замещает все вложенные узлы этого элемента или фрагмента новым единственным узлом `Text` с присвоенным свойству значением.

Методы

Node **appendChild**(Node *newChild*)

Этот метод добавляет узел *newChild* в документ, вставляя его в качестве последнего дочернего узла. Если узел *newChild* уже присутствует в дереве документа, он удаляется из дерева и вставляется в новое место. Если узел *newChild* является узлом `DocumentFragment`, сам узел не вставляется, а вместо этого в конец массива `childNodes[]` данного узла вставляются по порядку все дочерние узлы объекта `DocumentFragment`. Обратите внимание, что узел из другого документа (или созданный другим документом) не может быть вставлен в текущий документ. То есть свойство `ownerDocument` узла *newChild* должно совпадать со свойством `ownerDocument` данного узла. (См. `Document.adoptNode()`). Возвращает переданный ему узел.

Node **cloneNode**(boolean *deep*)

Создает и возвращает копию узла, для которого он вызван. Если в аргументе ему передается значение `true`, он также рекурсивно копирует всех потомков узла. В противном случае он копирует только данный узел, но не его дочерние узлы. Возвращаемый узел не является частью дерева документа, а его свойство `parentNode` имеет значение `null`. Когда копируется узел `Element`, то копируются и все его атрибуты. Однако следует отметить, что функции-обработчики событий, зарегистрированные для узла, не копируются.

unsigned short **compareDocumentPosition**(Node *other*)

Сравнивает позицию данного узла в документе с позицией узла *other* и возвращает число, биты которого описывают отношения между узлами. Если сравниваемые узлы являются одним и тем же узлом, все биты в результате будут сброшены, т. е. метод вернет 0. Иначе в возвращаемом значении будет установлен один или более битов. Константы DOCUMENT_POSITION_, перечисленные выше, являются символическими именами каждого из битов и имеют следующее значение:

DOCUMENT_POSITION_	**Значение**	**Описание**
DISCONNECTED	0x01	Два узла принадлежат разным документом, поэтому их позиции не могут сравниваться.
PRECEDING	0x02	Узел *other* располагается перед данным узлом.
FOLLOWING	0x04	Узел *other* располагается после данного узла.
CONTAINS	0x08	Узел *other* содержит данный узел. Когда установлен этот бит, всегда будет установлен бит PRECEDING.
CONTAINED_BY	0x10	Узел *other* содержится внутри данного узла. Когда установлен этот бит, всегда будет установлен бит FOLLOWING.

boolean **hasChildNodes**()

Возвращает true, если данный узел имеет один или более дочерних узлов, или false – в противном случае.

Node **insertBefore**(Node *newChild*, Node *refChild*)

Вставляет узел *newChild* в дерево документа как дочерний узел данного узла и возвращает вставленный узел. Новый узел позиционируется в массиве childNodes[] данного узла так, что он располагается непосредственно перед узлом *refChild*. Если аргумент *refChild* имеет значение null, узел *newChild* вставляется в конец массива childNodes[], как это делает метод appendChild(). Обратите внимание, что является ошибкой передавать в аргументе *refChild* узел, не являющийся дочерним по отношению к данному узлу.

Если узел *newChild* уже включен в дерево документа, он удаляется из дерева и затем вставляется в новую позицию. Если *newChild* является узлом фрагмента DocumentFragment, то в указанную позицию вставляется не сам узел, а все его дочерние узлы, в исходном порядке следования.

boolean **isDefaultNamespace**(string *namespace*)

Возвращает true, если URL-адрес пространства имен *namespace* совпадает с URL-адресом пространства имен по умолчанию, который возвращает вызов lookupNamespaceURI(null), и false – в противном случае.

boolean **isEqualNode**(Node *other*)

Возвращает true, если данный узел и узел *other* являются идентичными, т. е. имеют один и тот же тип, имя тега, атрибуты и (рекурсивно) дочерние узлы. Возвращает false, если два узла не являются эквивалентными.

boolean **isSameNode**(Node *other*)

Возвращает true, если данный узел и узел *other* являются одним и тем же узлом, и false – в противном случае. Вместо этого метода можно также просто использовать оператор ==.

`string` **`lookupNamespaceURI`**`(string` *`prefix`*`)`

Возвращает URL-адрес пространства имен, связанного с указанным префиксом пространства имен *prefix*, или `null`, если такой префикс не определен. Если аргумент *prefix* имеет значение `null`, возвращает URL-адрес пространства имен по умолчанию.

`string` **`lookupPrefix`**`(string` *`namespace`*`)`

Возвращает префикс пространства имен, связанного с указанным URL-адресом пространства имен, или `null`, если такое пространство имен не определено.

`void` **`normalize()`**

Нормализует все узлы, являющиеся потомками данного, объединяя смежные узлы и удаляя пустые. Обычно документы не имеют пустых или смежных текстовых узлов, но они могут появиться в результате добавления и удаления узлов сценарием.

`Node` **`removeChild`**`(Node` *`oldChild`*`)`

Этот метод удаляет дочерний узел *oldChild* из массива `childNodes[]` данного узла. Вызов этого метода с узлом, не являющимся дочерним, будет ошибкой. Метод `removeChild()` возвращает *oldChild* после его удаления. Старый дочерний узел *oldChild* продолжает быть действительным узлом и может быть позднее вставлен в документ.

`Node` **`replaceChild`**`(Node` *`newChild`*`, Node` *`oldChild`*`)`

Замещает узел *oldChild* в дереве документа другим узлом *newChild*. Узел *oldChild* должен быть дочерним для данного узла. Если *newChild* уже является частью документа, то он сначала удаляется из документа перед повторной вставкой в новую позицию. Если *newChild* является узлом фрагмента `DocumentFragment`, то вместо узла *newChild* в позицию, ранее занятую узлом *oldChild*, по порядку вставляются все его дочерние узлы.

NodeList

доступный только для чтения объект, подобный массиву, содержащий узлы

`NodeList` – это доступный только для чтения объект, подобный массиву, содержащий объекты `Node` (обычно элементы). Свойство `length` указывает, сколько узлов находится в списке; эти узлы можно извлекать, используя индексы от 0 до `length-1`. Вместо непосредственного индексирования объекта `NodeList` можно также использовать метод `item()`. Элементы `NodeList` всегда являются корректными объектами `Node`: объект `NodeList` никогда не содержит пустых (`null`) элементов.

Объектами `NodeList`, например, являются свойство `childNodes` и возвращаемые значения методов `Document.getElementsByTagName()`, `Element.getElementsByTagName()` и `HTMLDocument.getElementsByName()`. Поскольку объект `NodeList` является объектом, подобным массиву, в книге эти значения часто неформально называются массивами, например «массив `childNodes[]`».

Обратите внимание: объекты `NodeList` обычно являются «живыми»: они динамически отражают изменения в дереве документа. Например, если `NodeList` представляет дочерние узлы для указанного узла и вы удалите один из этих дочерних узлов, он будет удален и из вашего объекта `NodeList`. Будьте аккуратны при выполнении цикла по элементам `NodeList`, если тело цикла вносит изменения в дерево документа (например, удаляет узлы), которые могут влиять на содержимое `NodeList`!

Свойства

`readonly unsigned long` **`length`**

Количество узлов в объекте `NodeList`.

Методы

`Node` **`item`**`(unsigned long` *`index`*`)`

Возвращает узел в позиции *`index`* или `null`, если индекс *`index`* выходит за границы.

Option

элемент <option> в элементе Select **Node, Element**

Объект `Option` описывает вариант выбора внутри объекта `Select`. Свойства этого объекта определяют, выбран ли вариант по умолчанию или вариант, который выбран в данный момент, а также задают позицию, которую он занимает в массиве `options[]` содержащего его объекта `Select`, отображаемый им текст и значение, которое он передает на сервер при передаче данных родительской формы.

По историческим причинам элемент `Option` определяет конструктор, который можно использовать для создания и инициализации новых элементов `Option`. (Разумеется, можно также использовать обычный метод `Document.createElement()`.) После создания нового объекта `Option` его можно добавить в коллекцию `options` в объект `Select`. Дополнительные сведения приводятся в справочной статье `HTMLOptionsCollection`.

Конструктор

`new` **`Option`**`([string` *`text`*`, string` *`value`*`, boolean` *`defaultSelected`*`, boolean` *`selected`*`])`

Конструктор `Option()` создает новый элемент `<option>`. Четыре необязательных аргумента определяют значение свойства `textContent` (см. `Node`) элемента и начальные значения свойств `value`, `defaultSelected` и `selected`.

Свойства

`boolean` **`defaultSelected`**

Соответствует HTML-атрибуту `selected`. Определяет начальное значение состояния выбора данного варианта, а также значение, которое будет использоваться при сбросе формы в исходное состояние.

`boolean` **`disabled`**

Значение `true` означает, что данный элемент `<option>` недоступен. Варианты выбора становятся недоступными, если они или вмещающие их элементы `<optgroup>` имеют HTML-атрибут `disabled`.

`readonly Form` **`form`**

Элемент `<form>`, если имеется, содержащий данный элемент `Option`.

`readonly long` **`index`**

Индекс данного элемента `Option` в содержащем его элементе `Select`. (См. также `HTMLOptionsCollection`.)

`string` **`label`**

Значение HTML-атрибута `label`, если определен, иначе – значение свойства `textContent` (см. `Node`) данного элемента `Option`.

`boolean` **`selected`**

Имеет значение `true`, если данный вариант выбора выбран в настоящее время, или `false` – в противном случае.

string **text**

Значение свойства `textContent` (см. `Node`) данного элемента `Option`, из которого удалены начальные и завершающие пробельные символы, а каждые два или более смежных пробелов заменены одним символом пробела.

string **value**

Значение HTML-атрибута `value`, если определен, иначе – значение свойства `textContent`.

Output

элемент <output> HTML-форм **Node, Element, FormControl**

Объект `Output` представляет элемент `<output>` HTML-форм. В броузерах, поддерживающих их, объекты `Output` реализуют большинство свойств интерфейса `FormControl`.

Свойства

string **defaultValue**

Это свойство хранит начальное значение свойства `textContent` (см. `Node`) элемента `Output`. Когда выполняется сброс формы, свойство `value` элемента устанавливается в это значение. Если это свойство установлено и элемент `Output` отображает предыдущее значение свойства `defaultValue`, на экран будет выведено новое значение `defaultValue`. Иначе текущее отображаемое значение не изменится.

readonly DOMSettableTokenList **htmlFor**

HTML-атрибут `for` элемента `<output>` – это список атрибутов `id` элементов, разделенных пробелами, значения которых участвуют в вычислении содержимого, отображаемого элементом `<output>`. `for` является в языке JavaScript зарезервированным словом, поэтому соответствующее свойство называется `htmlFor`. Это свойство можно использовать, как если бы оно содержало обычную строку, или применять методы интерфейса `DOMTokenList` для чтения и изменения отдельных элементов списка.

PageTransitionEvent

объект события для событий «pageshow» и «pagehide» **Event**

Когда документ загружается впервые, вслед за событием «load» броузеры возбуждают событие «pageshow» и затем возбуждают событие «pageshow» всякий раз, когда страница восстанавливается из кэша в памяти. Обработчикам события «pageshow» передается объект `PageTransitionEvent`, свойство `persisted` которого имеет значение `true`, если страница была восстановлена из кэша, а не загружена из сети.

Объект `PageTransitionEvent` также передается обработчикам события «pagehide», но для событий «pagehide» свойство `persisted` объекта события всегда имеет значение `true`.

События «pageshow» и «pagehide» генерируются в объекте `Window`. Они не всплывают и не предусматривают действий по умолчанию, которые можно было бы отменить.

Свойства

readonly boolean **persisted**

Для события «pageshow» это свойство имеет значение `false`, если страница была загружена (или перезагружена) из сети или из дискового кэша. Оно имеет значение `true`, если страница была восстановлена из кэша в памяти.

Для события «pagehide» это свойство всегда имеет значение `true`.

PopStateEvent

событие перемещения по истории посещений **Event**

Веб-приложения, управляющие собственной историей посещений (раздел 22.2), используют метод `pushState()` объекта `History` для создания новых записей в истории и связывают с ними некоторое значение или объект, описывающие состояние приложения. Когда пользователь щелкает на кнопках броузера Back и Forward, выполняя переход между сохраненными состояниями, броузер генерирует события «popstate» в объекте `Window` и передает обработчику копию сохраненного состояния приложения в объекте события `PopStateEvent`.

Свойства

`readonly any` **`state`**

Это свойство хранит копию значения или объекта, описывающего состояние приложения, переданного методу `History.pushState()` или `History.replaceState()`. Состояние может быть любым значением, которое можно скопировать с использованием алгоритма структурированного копирования («Структурированные копии» в главе 22).

ProcessingInstruction

инструкция обработки в XML-документе **Node**

Этот редко используемый интерфейс представляет инструкцию обработки в XML-документе. Программисты, работающие с HTML-документами, никогда не столкнутся с узлом `ProcessingInstruction`.

Свойства

`string` **`data`**

Содержимое инструкции обработки (т. е. от первого непробельного символа после цели до закрывающих символов `?>`, но не включая их).

`readonly string` **`target`**

Цель инструкции обработки. Это первый идентификатор инструкции обработки, следующий за открывающими символами `<?`; он определяет «обработчик», для которого предназначена инструкция обработки.

Progress

индикатор хода выполнения операции **Node, Element**

Объект `Progress` представляет HTML-элемент `<progress>` и отображается как графический индикатор хода выполнения некоторой операции.

Когда заранее общий объем работы или времени на ее выполнение неизвестен, говорят, что элемент `Progress` находится в *неопределенном* состоянии. В таком состоянии он просто отображает некоторую «рабочую» анимацию, чтобы показать, что операция выполняется. Когда общий объем работы (в единицах времени или в байтах) известен заранее, элемент `Progress` находится в *определенном* состоянии и может отображать ход выполнения операции в процентах в виде некоторого графического представления.

Элемент `<progress>` введен спецификацией HTML5, поэтому на момент написания этих строк он был реализован не во всех броузерах.

Свойства

`readonly Form` **`form`**

Элемент `Form`, если имеется, являющийся предком для данного элемента или определяемый HTML-атрибутом `form`.

`readonly NodeList` **`labels`**

Объект, подобный массиву, содержащий элементы `Label`, связанные с этим элементом.

`double` **`max`**

Общий объем работы, который требуется выполнить. Например, при использовании элемента `Progress` для отображения хода операции выгрузки или загрузки, выполняемой объектом `XMLHttpRequest`, в это свойство можно записать общее количество байтов, которые требуется передать. Данное свойство соответствует HTML-атрибуту `max`. По умолчанию имеет значение 1.0.

`readonly double` **`position`**

Если элемент `Progress` находится в определенном состоянии, данное свойство содержит значение выражения `value/max`. Иначе оно будет иметь значение -1.

`double` **`value`**

Значение между 0 и `max`, определяющее уже выполненный объем работы. Это свойство соответствует HTML-атрибуту `value`. Если этот атрибут определен, элемент `Progress` находится в определенном состоянии. Если он отсутствует, элемент `Progress` находится в неопределенном состоянии. Чтобы переключиться из определенного в неопределенное состояние (например, потому что проигрыватель `MediaElement` получил событие «stalled»), можно воспользоваться методом `removeAttribute()` интерфейса `Element`.

ProgressEvent

событие продолжения загрузки, выгрузки или чтения файла **Event**

Все объекты – `ApplicationCache`, `FileReader` и `XMLHttpRequest` (версия 2) – возбуждают события, чтобы известить приложение о ходе выполнения операций передачи данных, таких как загрузка/выгрузка по сети или чтение файла. События этого рода известны как *события хода выполнения операции*, но только одно из них носит имя «progress». Другие события из этой категории, возбуждаемые объектами `FileReader` и `XMLHttpRequest`, – это события «loadstart», «load», «loadend», «error» и «abort».

Объект `XMLHttpRequest` также возбуждает событие «timeout». Объект `ApplicationCache` возбуждает несколько разных событий, имеющих отношение к ходу выполнения операции и описываемых здесь, но только одно из них носит имя «progress». Эти события возбуждаются в последовательности, которая начинается с события «loadstart» и всегда заканчивается событием «loadend». Непосредственно событию «loadend» предшествует событие «load», «error» или «abort», в зависимости от успеха выполнения операции. Между начальным «loadstart» и двумя заключительными событиями возбуждается ноль или более событий (с названием «progress»). (Объект `ApplicationCache` генерирует иную последовательность событий, но и он возбуждает событие «progress» в ходе обновления кэша, которое также относится к категории событий хода выполнения операции.)

Обработчики событий хода выполнения операции получают объект `ProgressEvent`, который определяет количество переданных байтов данных. Объект `ProgressEvent` никак не связан с HTML-элементом `<progress>`, описанным в справочной статье `Progress`, но объект `ProgressEvent`, передаваемый (например) обработчику `onprogress` объекта `XMLHttpRequest`, можно было бы использовать для обновления состояния элемента `<progress>`, обеспечивающего визуальное представление хода выполнения операции загрузки.

Свойства

`readonly boolean` **`lengthComputable`**

Имеет значение `true`, если известно общее количество байтов, предназначенных для передачи, и `false` – в противном случае. Если это свойство имеет значение `true`, процент выполнения операции для объекта е типа `ProgressEvent` можно вычислить как:

```
var percentComplete = Math.floor(100*e.loaded/e.total);
```

`readonly unsigned long` **`loaded`**

Количество уже переданных байтов.

`readonly unsigned long` **`total`**

Общее количество байтов, предназначенных для передачи, если известно, и 0 – в противном случае. Эту информацию можно получить, например, из свойства `size` объекта `Blob` или из заголовка `Content-Length`, возвращаемого веб-сервером.

Screen

предоставляет информацию о дисплее

Свойство `screen` любого объекта `Window` ссылается на объект `Screen`. Свойства этого глобального объекта содержат информацию об экране, на котором отображается броузер. JavaScript-программы могут руководствоваться этой информацией для оптимизации вывода в соответствии с возможностями дисплея пользователя. Например, программа может выбирать между большими и маленькими изображениями в зависимости от размера экрана.

Свойства

`readonly unsigned long` **`availHeight`**

Определяет доступную высоту экрана (в пикселах), на котором отображается веб-броузер. Эта доступная высота не включает пространство, занятое постоянно отображаемыми элементами рабочего стола, такими как панель задач в нижней части экрана.

`readonly unsigned long` **`availWidth`**

Определяет доступную ширину экрана в пикселах, на котором отображается веб-броузер. Эта доступная ширина не включает пространство, занимаемое постоянно отображаемыми элементами рабочего стола.

`readonly unsigned long` **`colorDepth`**

`readonly unsigned long` **`pixelDepth`**

Эти свойства, являющиеся синонимами, определяют глубину цвета в битах на пиксел.

`readonly unsigned long` **`height`**

Определяет общую высоту экрана в пикселах, на котором отображается веб-броузер. См. также `availHeight`.

`readonly unsigned long` **`width`**

Определяет общую ширину экрана в пикселах, на котором отображается веб-броузер. См. также `availWidth`.

Script

HTML-элемент <script> **Node, Element**

Объект `Script` представляет HTML-элемент `<script>`. Большинство его свойств просто соответствуют HTML-атрибутам с теми же именами, только при этом свойство `text` действует подобно свойству `textContent`, унаследованному от интерфейса `Node`.

Обратите внимание, что элемент `<script>` выполняется только один раз. Изменение свойства `src` или `text` существующего элемента `<script>` не приводит к запуску нового сценария. Однако эти свойства можно установить во вновь созданном элементе `<script>`, чтобы выполнить новый сценарий. Но имейте в виду, чтобы выполнить сценарий, тег `<script>` необходимо вставить в объект `Document`. Сценарий будет выполнен, когда будет установлено свойство `src` или `type` или когда он будет вставлен в документ, при выполнении обоих условий.

Свойства

`boolean` **async**

Имеет значение `true`, если элемент `<script>` имеет атрибут `async`, и `false` – в противном случае. Дополнительные сведения приводятся в разделе 13.3.1.

`string` **charset**

Кодировка символов в сценарии, на который ссылается свойство `src`. Обычно это свойство не определяется, и по умолчанию считается, что сценарий имеет ту же кодировку, что и вмещающий его документ.

`boolean` **defer**

Имеет значение `true`, если элемент `<script>` имеет атрибут `defer`, и `false` – в противном случае. Дополнительные сведения приводятся в разделе 13.3.1.

`string` **src**

URL-адрес сценария, который требуется загрузить.

`string` **text**

Текст между тегами `<script>` и `</script>`.

`string` **type**

MIME-тип с определением языка сценариев. По умолчанию устанавливается значение «text/javascript», благодаря чему для обычных сценариев на языке JavaScript не требуется устанавливать это свойство (или HTML-атрибут). При присваивании этому свойству собственного MIME-типа можно встраивать произвольные текстовые данные в элемент `<script>` для использования другими сценариями.

Select

графический список для выбора **Node, Element, FormControl**

Элемент `Select` представляет HTML-тег `<select>`, который отображается как графический список выбора. Если в определении HTML-элемента присутствует атрибут `multiple`, пользователь может одновременно выбрать в списке любое число вариантов. Если этот атрибут отсутствует, пользователь сможет выбрать только один вариант, и варианты ведут себя как радиокнопки – выбор одного из них приводит к отмене предыдущего выбора.

Варианты в элементе `Select` могут отображаться двумя различными способами. Во-первых, если атрибут `size` имеет значение больше 1 или если присутствует атрибут `multiple`, варианты отображаются в окне броузера в виде списка высотой `size` строк.

Если значение `size` меньше, чем число вариантов, в списке появится полоса прокрутки, чтобы обеспечить доступность всех вариантов. Во-вторых, если значение атрибута `size` равно 1 и атрибут `multiple` не указан, текущий выбранный вариант отображается в единственной строке, а список всех остальных вариантов доступен через раскрывающееся меню. Первый стиль представления позволяет видеть все доступные варианты, но занимает больше пространства в окне броузера. Второй стиль требует минимум пространства, но не дает возможности увидеть альтернативные варианты все сразу. По умолчанию свойство `size` получает значение 4 при наличии атрибута `multiple` и 1 – в противном случае.

Самый большой интерес представляет свойство `options[]` элемента `Select`. Это объект, подобный массиву, содержащий элементы `<option>` (см. `Option`), которые описывают варианты выбора, представленные в элементе `Select`. По историческим причинам этот объект, подобный массиву, имеет необычные особенности, касающиеся выполнения операций добавления и удаления элементов `<option>`. Дополнительные сведения приводятся в статье `HTMLOptionsCollection`.

Если в элементе `Select` отсутствует атрибут `multiple`, определить, какой вариант выбран, можно с помощью свойства `selectedIndex`. Однако если допускается возможность одновременного выбора нескольких вариантов, это свойство содержит индекс первого выбранного варианта. Чтобы определить все множество выбранных вариантов, необходимо обойти в цикле массив `options[]` и проверить свойство `selected` каждого объекта `Option`.

Свойства

В дополнение к свойствам, перечисленным ниже, элементы `Select` также поддерживают свойства интерфейса `Element` и `FormControl` и имеют свойства `multiple`, `required` и `size`, соответствующие HTML-атрибутам.

`unsigned long` **`length`**

Количество элементов в коллекции `options`. Объекты `Select` сами являются объектами, подобными массивам, поэтому для объекта `s` типа `Select` и числа `n` выражение `s[n]` возвращает то же значение, что и `s.options[n]`.

`readonly HTMLOptionsCollection` **`options`**

Объект, подобный массиву, с элементами `Option`, содержащимися в данном элементе `Select`. Описание исторически сложившегося поведения этой коллекции приводится в справочной статье `HTMLOptionsCollection`.

`long` **`selectedIndex`**

Индекс выбранного варианта в массиве `options`. Если ни один из вариантов не выбран, значение этого свойства равно -1. Если выбрано более одного варианта, свойство `selectedIndex` определяет индекс только первого из них.

Установка значения этого свойства приводит к выбору указанного варианта и отменяет выбор всех остальных, даже если в объекте `Select` указан атрибут `multiple`. Если выбор реализован в виде списка (когда значение свойства `size` > 1), то выбор всех вариантов можно отменить, установив свойство `selectedIndex` равным -1. Обратите внимание: этот способ изменения выбора не приводит к вызову обработчика события `onchange()`.

`readonly HTMLCollection` **`selectedOptions`**

Доступный только для чтения объект, подобный массиву, содержащий выбранные элементы `Option`. Это новое свойство, определяемое спецификацией HTML5, которое на момент написания этих строк было реализовано не во всех броузерах.

Методы

Все методы, перечисленные ниже, делегируют выполнение операций одноименным методам свойства `options`; дополнительные сведения приводятся в справочной статье `HTMLOptionsCollection`. В дополнение к этим методам элементы `Select` реализуют методы интерфейсов `Element` и `FormControl`.

`void` **`add`**`(Element` *`element`*`, [any` *`before`*`])`

Действует подобно методу `options.add()`, добавляя новый элемент `Option`.

`any` **`item`**`(unsigned long` *`index`*`)`

Действует подобно методу `options.item()` и возвращает элемент `Option`. Он также неявно вызывается, когда пользователь обращается к элементу `Select` как к массиву.

`any` **`namedItem`**`(string` *`name`*`)`

Действует подобно методу `options.namedItem()`. См. `HTMLOptionsCollection`.

`void` **`remove`**`(long` *`index`*`)`

Действует подобно методу `options.remove()`, удаляя элемент Option. См. `HTMLOptionsCollection`.

Storage

хранилище пар имя/значение на стороне клиента

Свойства `localStorage` и `sessionStorage` объекта `Window` являются объектами `Storage`, которые представляют хранимые на стороне клиента ассоциативные массивы, отображающие строковые ключи в значения. Теоретически объект `Storage` может хранить любые значения, которые можно копировать с применением алгоритма структурированного копирования (врезка «Структурированные копии» в главе 22). Однако на момент написания данных строк броузеры позволяли сохранять только строковые значения.

Методы объекта `Storage` позволяют добавлять новые пары ключ/значение, удалять их и получать значение, связанное с определенным ключом. Однако нет необходимости явно вызывать эти методы: вместо них можно использовать операцию индексирования и оператор `delete` и обрабатывать свойства `localStorage` и `sessionStorage`, как если бы они были обычными объектами.

При изменении содержимого объекта `Storage` любые другие объекты `Window`, имеющие доступ к той же области хранилища (т. е. отображающие документы с тем же происхождением), будут извещены об изменениях с помощью объекта события `StorageEvent`.

Свойства

`readonly unsigned long` **`length`**

Количество хранящихся пар ключ/значение.

Методы

`void` **`clear`**`()`

Удаляет все хранящиеся пары ключ/значение.

`any` **`getItem`**`(string` *`key`*`)`

Возвращает значение, связанное с ключом *key*. (В текущих, на момент написания этих строк, реализациях всегда возвращалась строка.) Этот метод вызывается неявно при индексировании объекта `Storage`, с целью получить значение свойства с именем в аргументе *key*.

`string` **`key`**`(unsigned long` *`n`*`)`

Возвращает ключ с индексом *n*, хранящийся в данном объекте `Storage`, или `null`, если *n* больше или равно `length`. Обратите внимание, что порядок следования ключей может изменяться при добавлении и удалении пар ключ/значение.

`void` **`removeItem`**`(string` *`key`*`)`

Удаляет из объекта `Storage` ключ *key* и связанное с ним значение. Этот метод вызывается неявно при использовании оператора `delete`, с целью удалить свойство с именем в аргументе *key*.

`void` **`setItem`**`(string` *`key`*`, any` *`value`*`)`

Добавляет ключ *key* и значение *value* в данный объект `Storage`, замещая значение, прежде связанное с этим ключом *key*. Этот метод вызывается неявно при присваивании значения свойству объекта `Storage` с именем в аргументе *key*. То есть вместо явного вызова метода `setItem()` можно использовать обычную операцию присваивания значения свойству.

StorageEvent

Event

Свойства `localStorage` и `sessionStorage` объекта `Window` ссылаются на объекты `Storage`, представляющие хранилища на стороне клиента (раздел 20.1). Если имеется несколько окон, вкладок или фреймов, отображающих документы с общим происхождением, все они будут иметь доступ к одному и тому же хранилищу. Если сценарий в одном окне изменит содержимое хранилища, во всех других объектах Window, имеющих доступ к этому хранилищу, будет сгенерировано событие «storage». (Обратите внимание, что это событие не генерируется в окне, в котором были выполнены изменения.) События «storage» генерируются в объекте `Window` и не всплывают. Для них не предусматривается действий по умолчанию, которые можно было бы отменить. Обработчикам события «storage» передается объект события `StorageEvent`, свойства которого описывают изменения, внесенные в хранилище.

Свойства

`readonly string` **`key`**

Это свойство хранит ключ, который был установлен или удален. Если все хранилище было очищено вызовом метода `Storage.clear()`, это свойство (а также свойства `newValue` и `oldValue`) будет иметь значение `null`.

`readonly any` **`newValue`**

Новое значение ключа `key`. Будет иметь значение `null` при удалении ключа. На момент написания этих строк броузеры позволяли сохранять только строковые значения.

`readonly any` **`oldValue`**

Старое значение изменившегося ключа `key`. Будет иметь значение `null` при добавлении нового ключа. На момент написания этих строк броузеры позволяли сохранять только строковые значения.

`readonly Storage` **`storageArea`**

Это свойство будет содержать то же значение, что и свойство `localStorage` или `sessionStorage` объекта `Window`, принявшего это событие, и указывает, содержимое какого хранилища изменилось.

`readonly string` **`url`**

URL-адрес документа, сценарий которого внес изменения в хранилище.

Style

HTML-элемент <style> **Node, Element**

Объект `Style` представляет HTML-тег `<style>`.

Свойства

`boolean` **`disabled`**

Установка этого свойства в значение `true` отключит таблицу стилей, связанную с данным элементом `<style>`, а установка в значение `false` снова включит ее.

`string` **`media`**

Это свойство соответствует HTML-атрибуту `media` и определяет устройства, при отображении в которых должна применяться указанная таблица стилей.

`boolean` **`scoped`**

Имеет значение `true`, если в элементе `<style>` присутствует HTML-атрибут `scoped`, и `false` – в противном случае. На момент написания этих строк броузеры не поддерживали контекстные (scoped) таблицы стилей.

`readonly CSSStyleSheet` **`sheet`**

Объект `CSSStyleSheet`, определяемый данным элементом `<style>`.

`string` **`title`**

Все HTML-элементы имеют атрибут `title`. С помощью этого атрибута или свойства элемента `<style>` можно дать пользователю возможность выбрать альтернативную таблицу стилей по названию, и указанное значение свойства `title` может отображаться в интерфейсе броузера в некотором оформлении.

`string` **`type`**

Соответствует HTML-атрибуту `type`. По умолчанию имеет значение «text/css», и обычно нет необходимости указывать другое значение этого атрибута.

Table

HTML-элемент <table> **Node, Element**

Объект `Table` представляет HTML-элемент `<table>` и определяет несколько удобных свойств и методов для получения и модификации различных частей таблицы. Эти методы и свойства облегчают работу с таблицами, но они также могут быть продублированы с помощью базовых DOM-методов.

HTML-таблицы конструируются из разделов, строк и ячеек. См. также `TableCell`, `TableRow` и `TableSection`.

Свойства

В дополнение к свойствам, перечисленным ниже, элементы `Table` имеют также свойство `summary`, соответствующее HTML-атрибуту с тем же именем.

`Element` **`caption`**

Ссылка на элемент `<caption>` в таблице или `null`, если он отсутствует.

readonly HTMLCollection **rows**

Объект, подобный массиву, содержащий объекты TableRow, который представляет все строки в таблице. Включает все строки, определяемые внутри тегов <thead>, <tfoot> и <tbody>.

readonly HTMLCollection **tBodies**

Объект, подобный массиву, содержащий объекты TableSection, который представляет все разделы <tbody> в таблице.

TableSection **tFoot**

Элемент <tfoot> таблицы или null, если он отсутствует.

TableSection **tHead**

Элемент <thead> таблицы или null, если он отсутствует.

Методы

Element **createCaption()**

Возвращает объект Element, представляющий элемент <caption> таблицы. Если в таблице уже имеется элемент <caption>, метод просто вернет его. Если в таблице отсутствует элемент <caption>, этот метод создаст новый (пустой) элемент, вставит его в таблицу и вернет вызывающей программе.

TableSection **createTBody()**

Создаст новый элемент <tbody>, вставит в таблицу и вернет его. Новый элемент вставляется после последнего элемента <tbody> в таблице или в конец таблицы.

TableSection **createTFoot()**

Возвращает объект TableSection, представляющий первый элемент <tfoot> в таблице. Если в таблице уже есть нижний колонтитул, метод просто вернет его. Если таблица не имеет нижнего колонтитула, этот метод создаст новый (пустой) элемент <tfoot>, вставит его в таблицу и вернет вызывающей программе.

TableSection **createTHead()**

Возвращает объект TableSection, представляющий первый элемент <thead> в таблице. Если в таблице уже имеется заголовок, метод просто вернет его. Если таблица не имеет заголовка, этот метод создаст новый (пустой) элемент <thead>, вставит его в таблицу и вернет вызывающей программе.

void **deleteCaption()**

Удаляет из таблицы первый элемент <caption>, если он существует.

void **deleteRow(**long *index***)**

Удаляет из таблицы строку с индексом *index*. Строки нумеруются в порядке, в каком они следуют в исходном документе. Строки в разделах <thead> и <tfoot> нумеруются вместе со всеми остальными строками в таблице.

void **deleteTFoot()**

Удаляет из таблицы первый элемент <tfoot>, если он существует.

void **deleteTHead()**

Удаляет из таблицы первый элемент <thead>, если он существует.

TableRow **insertRow(**[long *index*]**)**

Создает новый элемент <tr>, вставляет в таблицу в позицию, определяемую аргументом *index*, и возвращает его.

Новая строка вставляется в том же разделе таблицы и непосредственно перед существующей строкой, в позиции, заданной аргументом *index*. Если значение аргумента *index* равно количеству строк в таблице (или -1), новая строка добавляется в конец последнего раздела таблицы. Если таблица изначально пуста, новая строка вставляется в новый раздел `<tbody>`, который в свою очередь вставляется в таблицу.

Для добавления содержимого в только что созданную строку можно использовать вспомогательный метод `TableRow.insertCell()`. См. также описание метода `insertRow()` объекта `TableSection`.

TableCell

ячейка в HTML-таблице **Node, Element**

Объект `TableCell` представляет элементы `<td>` и `<th>`.

Свойства

`readonly long` **`cellIndex`**

Позиция данной ячейки внутри строки.

`unsigned long` **`colSpan`**

Значение HTML-атрибута `colspan` в виде числа.

`unsigned long` **`rowSpan`**

Значение HTML-атрибута `rowspan` в виде числа.

TableRow

элемент `<tr>` в HTML-таблице **Node, Element**

Объект `TableRow` представляет строку (элемент `<tr>`) в HTML-таблице, а также определяет свойства и методы для работы с элементами `TableCell`, содержащимися в строке.

Свойства

`readonly HTMLCollection` **`cells`**

Объект, подобный массиву, содержащий объекты `TableCell`, представляющие элементы `<td>` и `<th>` в данной строке.

`readonly long` **`rowIndex`**

Индекс этой строки в таблице.

`readonly long` **`sectionRowIndex`**

Позиция этой строки в данном разделе (т.е. внутри данного элемента `<thead>`, `<tbody>` или `<tfoot>`).

Методы

`void` **`deleteCell`**`(long` *`index`*`)`

Удаляет ячейку в позиции *index* в строке таблицы.

`Element` **`insertCell`**`([long` *`index`*`])`

Создает новый элемент `<td>`, вставляет в строку в указанную позицию и возвращает его. Новая ячейка вставляется непосредственно перед ячейкой, находящейся в данный момент в позиции, определяемой аргументом *index*. Если аргумент *index* отсутствует, равен количеству ячеек в строке или -1, новая ячейка добавляется в конец строки.

Обратите внимание: этот вспомогательный метод позволяет вставлять только ячейки данных `<td>`. Чтобы вставить ячейку в строку верхнего колонтитула, необходимо создать и вставить элемент `<th>` методами `Document.createElement()` и `Node.insertBefore()` или другими родственными им методами.

TableSection

раздел верхнего или нижнего колонтитула либо тела таблицы **Node, Element**

Интерфейс `TableSection` представляет раздел `<tbody>`, `<thead>` или `<tfoot>` HTML-таблицы. Свойства `tHead` и `tFoot` объектов `Table` являются объектами `TableSection`, а свойство `tBodies` – коллекцией `HTMLCollection` объектов `TableSection`.

Объект `TableSection` содержит объекты `TableRow` и сам содержится в объекте `Table`.

Свойства

`readonly HTMLCollection` **`rows`**

Объект, подобный массиву, содержащий объекты `TableRow`, представляющие строки в этом разделе таблицы.

Методы

`void` **`deleteRow(`**`long` *`index`***`)`**

Удаляет строку в указанной позиции в данном разделе.

`TableRow` **`insertRow(`**`[long` *`index`*`]`**`)`**

Создает новый элемент `<tr>`, вставляет в данный раздел таблицы в указанную позицию и возвращает его. Если аргумент *`index`* опущен, равен количеству строк в разделе или -1, новая строка добавляется в конец раздела. В противном случае новая строка вставляется непосредственно перед строкой, находящейся в данный момент в позиции, заданной аргументом *`index`*. Обратите внимание: для этого метода аргумент *`index`* определяет позицию строки внутри одного раздела, а не в таблице в целом.

Text

текстовая последовательность в документе **Node**

Узел `Text` представляет обычный текст в документе и обычно располагается в дереве документа в виде дочернего узла по отношению к узлу `Element`. Текстовое содержимое узла `Text` доступно через свойство `data` или через свойства `nodeValue` и `textContent`, унаследованные от `Node`. Создать новый узел `Text` можно с помощью `Document.createTextNode()`. Текстовые узлы никогда не имеют дочерних узлов.

Свойства

`string` **`data`**

Текст, содержащийся в данном узле.

`readonly unsigned long` **`length`**

Длина текста в символах.

`readonly string` **`wholeText`**

Текстовое содержимое данного узла и любых смежных с ним текстовых узлов, предшествующих ему и следующих за ним. После вызова метода `normalize()` родительского элемента `Node` это свойство будет иметь то же значение, что и свойство `data`.

Методы

Эти методы вам не придется использовать на практике, если только вы не соберетесь написать текстовый редактор с веб-интерфейсом.

void **appendData**(string *text*)

Добавляет текст *text* **в конец данного текстового узла.**

void **deleteData**(unsigned long *offset*, unsigned long *count*)

Удаляет *count* **символов из данного текстового узла, начиная с символа в позиции** *offset*. **Если сумма значений** *offset* **и** *count* **превысит количество символов в текстовом узле, будут удалены все символы до конца строки, начиная с символа в позиции** *offset*.

void **insertData**(unsigned long *offset*, string *text*)

Вставляет текст *text* **в текстовый узел в позицию** *offset*.

void **replaceData**(unsigned long *offset*, unsigned long *count*, string *text*)

Замещает *count* **символов, начиная с позиции** *offset*, **содержимым строки** *text*. **Если сумма значений** *offset* **и** *count* **превысит значение свойства** length **текстового узла, будут замещены все символы, начиная с позиции** *offset*.

Text **replaceWholeText**(string *text*)

Создает новый узел Text, **содержащий текст** *text*, **а затем замещает данный и смежные с ним текстовые узлы новым узлом и возвращает новый узел. См. также описание свойства** wholeText **выше и метода** normalize() **интерфейса** Node.

Text **splitText**(unsigned long *offset*)

Разбивает узел Text **на два по смещению** *offset*. **Исходный узел** Text **модифицируется так, чтобы он содержал весь текст до символа в позиции** *offset*, **но не включая его. Создается новый узел, который содержит все символы от позиции** *offset* **(включая ее) до конца строки. Этот новый узел** Text **является возвращаемым значением метода. Кроме того, если исходный узел** Text **имеет родительский узел, то новый узел вставляется в родительский узел непосредственно после исходного узла.**

string **substringData**(unsigned long *offset*, unsigned long *count*)

Извлекает и возвращает подстроку длиной *count* **символов, начинающуюся с символа в позиции** *offset* **в тексте узла** Text. **Если узел** Text **содержит слишком большой объем текста, этот метод может оказаться более эффективным, чем метод** String.substring().

TextArea

многострочная область ввода текста — **Node, Element, FormControl**

Объект TextArea **представляет HTML-элемент** <textarea> **– многострочное текстовое поле ввода, часто используемое в HTML-формах. Начальное содержимое текстовой области вставляется между тегами** <textarea> **и** </textarea>. **Получить и изменить текст можно с помощью свойства** value.

Объект TextArea **– это элемент ввода формы, подобный элементам** Input **и** Select. **Аналогично этим объектам он определяет свойства** form, name, type **и** value, **а также другие свойства и методы, описанные в справочной статье** FormControl.

Свойства

В дополнение к свойствам, перечисленным ниже, элементы TextArea **определяют свойства интерфейсов** Element **и** FormControl, **а также следующие свойства, соответствующие HTML-атрибутам:** cols, maxLength, rows, placeholder, readOnly, required **и** wrap.

`string` **`defaultValue`**

Начальное текстовое содержимое элемента `<textarea>`. Когда выполняется сброс формы, содержимое текстовой области восстанавливается в это значение. Это свойство имеет то же значение, что и свойство `textContent`, унаследованное от `Node`.

`unsigned long` **`selectionEnd`**

Возвращает или устанавливает индекс первого введенного символа, следующего за выделенным текстом. См. также `setSelectionRange()`.

`unsigned long` **`selectionStart`**

Возвращает или устанавливает индекс первого выделенного символа в элементе `<textarea>`. См. также `setSelectionRange()`.

`readonly unsigned long` **`textLength`**

Длина свойства `value` в символах (см. `FormControl`).

Методы

В дополнение к методам, перечисленным ниже, элементы `TextArea` реализуют методы интерфейсов `Element` и `FormControl`.

`void` **`select()`**

Выделяет весь текст в элементе `<textarea>`. Во многих броузерах это означает, что текст будет выделен цветом и при вводе очередного символа выделенный текст будет удален и замещен введенным символом.

`void` **`setSelectionRange(`**`unsigned long` *`start`*, `unsigned long` *`end`***`)`**

Выделяет текст в элементе `<textarea>`, начиная с символа в позиции *`start`* и заканчивая (но не включая его) символом в позиции *`end`*.

TextMetrics

определяет размеры текстовой строки

Объект `TextMetrics` возвращается методом `measureText()` объекта `CanvasRenderingContext2D`. Его свойство `width` хранит ширину текста в CSS-пикселах. В будущем могут быть добавлены дополнительные размеры.

Свойства

`readonly double` **`width`**

Ширина текста в CSS-пикселах.

TimeRanges

множество фрагментов мультимедийных данных

Свойства `buffered`, `played` и `seekable` элемента `MediaElement` представляют блоки мультимедийных данных, загруженных в буфер, которые были проиграны и которые можно начать проигрывать. Каждый из этих блоков может включать множество разрозненных фрагментов (это характерно для свойства `played`, когда, например, пользователь перепрыгивает к середине видеозаписи). Объект `TimeRanges` представляет ноль или более разрозненных фрагментов. Свойство `length` определяет количество фрагментов, а методы `start()` и `end()` возвращают границы каждого фрагмента.

Объекты `TimeRanges`, возвращаемые объектами `MediaElement`, всегда *нормализованы*, т. е. в них отсутствуют пустые и смежные или перекрывающиеся фрагменты.

Свойства

`readonly unsigned long` **`length`**

Количество фрагментов, представленных данным объектом `TimeRanges`.

Методы

`double` **`end`**`(unsigned long` *`n`*`)`

Возвращает конец фрагмента *n* (в секундах) или возбуждает исключение, если значение *n* меньше нуля или больше или равно значению свойства `length`.

`double` **`start`**`(unsigned long` *`n`*`)`

Возвращает начало фрагмента *n* (в секундах) или возбуждает исключение, если значение *n* меньше нуля или больше или равно значению свойства `length`.

TypedArray[1]

массивы с двоичными элементами фиксированного размера **ArrayBufferView**

Типизированные массивы являются подтипами `ArrayBufferView`, который интерпретирует байты в объекте `ArrayBuffer`, на котором он основан, как массив чисел и позволяет читать и изменять элементы этого массива. Данная справочная статья описывает не какой-то конкретный тип типизированных массивов, а охватывает восемь разных видов *типизированных массивов*. Все эти восемь типов являются подтипами `ArrayBufferView` и отличаются друг от друга только количеством байтов, выделенных для одного элемента массива и способом интерпретации этих элементов. В число этих восьми типов входят:

`Int8Array`

Массив однобайтных элементов, которые интерпретируются как целые со знаком.

`Int16Array`

Массив двухбайтных элементов, которые интерпретируются как целые со знаком, с использованием порядка следования байтов, определяемого платформой.

`Int32Array`

Массив четырехбайтных элементов, которые интерпретируются как целые со знаком, с использованием порядка следования байтов, определяемого платформой.

`Uint8Array`

Массив однобайтных элементов, которые интерпретируются как целые без знака.

`Uint16Array`

Массив двухбайтных элементов, которые интерпретируются как целые без знака, с использованием порядка следования байтов, определяемого платформой.

`Uint32Array`

Массив четырехбайтных элементов, которые интерпретируются как целые без знака, с использованием порядка следования байтов, определяемого платформой.

`Float32Array`

Массив четырехбайтных элементов, которые интерпретируются как вещественные числа, с использованием порядка следования байтов, определяемого платформой.

---

[1] В клиентском JavaScript нет типа TypedArray. Автор использовал это символическое имя для более краткого обозначения типизированных массивов. – *Прим. ред.*

`Float64Array`

Массив восьмибайтных элементов, которые интерпретируются как вещественные числа, с использованием порядка следования байтов, определяемого платформой.

Как следует из названий, эти типы являются объектами, подобными массивам, которые обеспечивают доступ к значениям элементов с использованием привычной формы обращения к массивам с квадратными скобками. Отметьте однако, что объекты этих типов всегда имеют фиксированную длину.

Как отмечалось в описании выше, классы `TypedArray` по умолчанию используют порядок следования байтов, определяемый платформой. См. описание типа `DataView`, предназначенного для представления `ArrayBuffer`, который позволяет явно определять порядок следования байтов.

Конструктор

new *TypedArray*(unsigned long *length*)
new *TypedArray*(TypedArray *array*)
new *TypedArray*(type[] *array*)
new *TypedArray*(ArrayBuffer *buffer*, [unsigned long *byteOffset*], [unsigned long *length*])

Для каждой из восьми разновидностей типизированных массивов имеется конструктор, который можно вызвать одним из приведенных выше четырех способов. Конструкторы действуют следующим образом:

- Если конструктор вызывается с единственным числовым аргументом, он создает новый типизированный массив с указанным количеством элементов и инициализирует каждый элемент нулем.
- Если конструктору передается единственный объект типизированного массива, он создает новый типизированный массив с тем же количеством элементов, что и в массиве в аргументе, и копирует элементы из массива в аргументе во вновь созданный массив. Тип массива в аргументе не обязательно должен совпадать с типом создаваемого массива.
- Если конструктору передается единственный массив (истинный массив), он создает новый типизированный массив с тем же количеством элементов, что и в массиве в аргументе, и копирует элементы из массива в аргументе во вновь созданный массив.
- Наконец, если конструктору передается объект `ArrayBuffer` с необязательными аргументами, определяющими смещение и длину, он создает новый типизированный массив, который является представлением указанной области объекта `ArrayBuffer`. Длина нового типизированного массива зависит от размера области в `ArrayBuffer` и размера элементов в типизированном массиве.

Константы

long **BYTES_PER_ELEMENT**

Количество байтов, занимаемых каждым элементом данного массива в лежащем в основе объекте `ArrayBuffer`. Эта константа будет иметь значение 1, 2, 4 или 8, в зависимости от вида типизированного массива.

Свойства

readonly unsigned long **length**

Количество элементов в массиве. Типизированные массивы имеют фиксированный размер, поэтому значение этого свойства никогда не изменяется. Не путайте это свойство со свойством `byteLength`, унаследованным от `ArrayBufferView`.

Методы

`void` **`set(`**`TypedArray array, [unsigned long offset]`**`)`**

Копирует элементы массива *array* в данный типизированный массив, начиная с индекса *offset*.

`void` **`set(`**`number[] array, [unsigned long offset]`**`)`**

Эта версия метода `set()` подобна предыдущей, но принимает не типизированный, а истинный массив.

TypedArray **`subarray(`**`long start, long end`**`)`**

Возвращает новый типизированный массив, опирающийся на тот же объект `ArrayBuffer`, что и данный массив. Первым элементом возвращаемого массива является элемент данного массива с индексом *start*. А последним – элемент данного массива с индексом *end*-1. Отрицательные значения в аргументах *start* и *end* интерпретируются как смещения относительно конца данного массива.

URL

методы для работы с URL-адресами, ссылающимися на объекты Blob

Свойство `URL` объекта `Window` ссылается на этот объект `URL`. В будущем этот объект может превратиться в конструктор класса, реализующего средства синтаксического анализа и выполнения операций с URL-адресами. Однако на момент написания этих строк оно служило пространством имен для описываемых ниже двух функций, предназначенных для работы с URL-адресами, ссылающимися на объекты. Дополнительные сведения об объектах `Blob` и об URL-адресах, ссылающихся на них, приводятся в разделах 22.6 и 22.6.4.

Объект `URL` был новинкой на момент написания этих строк, и его прикладной интерфейс еще не был стабилизирован. Вам может потребоваться использовать префикс производителя броузера при работе с ним, например `webkitURL`.

Функция

`string` **`createObjectURL(`**`Blob blob`**`)`**

Возвращает URL-адрес, ссылающийся на двоичный объект *blob*. HTTP GET-запросы по этому URL-адресу будут возвращать содержимое *blob*.

`void` **`revokeObjectURL(`**`string url`**`)`**

Отзывает (делает недействительным) адрес *url* так, что он больше не будет связан с каким-либо объектом `Blob` и не сможет использоваться для загрузки данных.

Video

HTML-элемент <video> **Node, Element, MediaElement**

Объект `Video` представляет HTML-элемент `<video>`. Элементы `<video>` и `<audio>` очень похожи друг на друга, и их общие свойства и методы были описаны в справочной статье `MediaElement`. Эта справочная статья описывает несколько дополнительных свойств, которыми обладают только объекты `Video`.

Свойства

`DOMSettableTokenList` **`audio`**

Это свойство определяет аудиопараметры видеозаписи. Параметры указываются в HTML-атрибуте `audio` в виде списка названий параметров, разделенных пробелами, и в программном коде на языке JavaScript отражаются в множество `DOMSet-`

`tableTokenList`. Однако на момент написания этих строк стандарт HTML5 определял только один параметр («muted»), поэтому данное свойство можно интерпретировать как строку.

`unsigned long` **`height`**

Высота элемента `<video>` на экране в CSS-пикселах. Соответствует HTML-атрибуту `height`.

`string` **`poster`**

URL-адрес изображения, отображаемого в качестве «афиши» до того, как будет запущено проигрывание видеозаписи. Соответствует HTML-атрибуту `poster`.

`readonly unsigned long` **`videoHeight`**
`readonly unsigned long` **`videoWidth`**

Эти свойства возвращают истинную высоту и ширину кадра видеозаписи в CSS-пикселах. Эти свойства будут иметь нулевые значения, пока элемент `<video>` не загрузит метаданные (пока свойство `readyState` имеет значение `HAVE_NOTHING`, и не было сгенерировано событие «loadedmetadata»).

`unsigned long` **`width`**

Желаемая ширина элемента `<video>` на экране в CSS-пикселах. Соответствует HTML-атрибуту `width`.

WebSocket

двунаправленное сетевое соединение, подобное сокету **EventTarget**

Объект `WebSocket` представляет долгоживущее, двунаправленное сетевое соединение с сервером, поддерживающим протокол веб-сокетов. Данная модель сетевых взаимодействий существенно отличается от модели запрос/ответ, которую реализует протокол HTTP. Создать новое сетевое соединение можно вызовом конструктора `WebSocket()`. Отправлять текстовые данные на сервер можно с помощью метода `send()`, а принимать сообщения от сервера – с помощью обработчика событий «message». Дополнительные сведения приводятся в разделе 22.9.

Веб-сокеты – это новый прикладной интерфейс для веб-приложений; на момент написания этих строк поддерживался не всеми броузерами.

Конструктор

`new` **`WebSocket`**`(string` *url*, `[string[]` *protocols*`])`

Конструктор `WebSocket()` создает новый объект `WebSocket` и запускает (асинхронный) процесс установления соединения с сервером, поддерживающим протокол веб-сокетов. Аргумент *url* определяет сервер, с которым требуется установить соединение, и должен быть абсолютным URL-адресом с URL-схемой `ws://` или `wss://`. Аргумент *protocols* – это массив названий подпротоколов. С помощью этого аргумента клиент может сообщить серверу, какие версии протоколов он поддерживает. Сервер должен выбрать один из них и информировать клиента о своем выборе в процессе установления соединения. В аргументе *protocols* можно также передать строку, а не массив: в этом случае значение аргумента будет интерпретироваться как массив с единственным элементом.

Константы

Следующие константы определяют допустимые значения свойства `readyState`.

`unsigned short` **`CONNECTING = 0`**

Продолжается выполнение процедуры установления соединения.

unsigned short **OPEN** = 1

Объект `WebSocket` соединен с сервером; можно отправлять и принимать сообщения.

unsigned short **CLOSING** = 2

Соединение закрывается.

unsigned short **CLOSED** = 3

Соединение закрыто.

Свойства

readonly unsigned long **bufferedAmount**

Количество символов сообщения, переданного методу `send()`, которые еще не были отправлены фактически. Это свойство можно использовать при передаче больших объемов данных, чтобы гарантировать, что программа не будет отправлять сообщения быстрее, чем они могут быть переданы по сети.

readonly string **protocol**

Если конструктору `WebSocket()` был передан массив подпротоколов, это свойство будет хранить один из них, выбранный сервером. Обратите внимание, что в первый момент после создания объекта `WebSocket` соединение еще не установлено и выбор сервера неизвестен, поэтому первоначально это свойство содержит пустую строку. Если конструктору был передан список протоколов, значение этого свойства изменится в соответствии с выбором сервера, когда будет сгенерировано событие «open».

readonly unsigned short **readyState**

Текущее состояние соединения. Значением этого свойства может быть одна из констант, перечисленных выше.

readonly string **url**

Это свойство хранит URL-адрес, который был передан конструктору `WebSocket()`.

Методы

void **close()**

Если соединение еще не закрыто или для него еще не была запущена процедура закрытия, этот метод инициирует процесс его закрытия и присваивает свойству `readyState` значение `CLOSING`. События «message» могут продолжать возбуждаться даже после вызова метода `close()`, пока свойство `readyState` не получит значение `CLOSED` и не будет возбуждено событие «close».

void **send**(string *data*)

Отправляет данные *data* на сервер, подключенный к другому концу соединения. Этот метод возбуждает исключение, когда вызывается до того, как будет сгенерировано событие «open», т.е. пока свойство `readyState` имеет значение `CONNECTING`. Протокол веб-сокетов поддерживает обмен двоичными данными, но на момент написания этих строк текущая версия прикладного интерфейса веб-сокетов поддерживала только текстовые сообщения.

Обработчики событий

Сетевые взаимодействия по своей природе являются асинхронными, и, подобно объекту `XMLHttpRequest`, объект `WebSocket` также опирается на использование событий. Он определяет четыре свойства регистрации обработчиков событий, а также реализует интерфейс `EventTarget`, благодаря чему обработчики событий можно также регистрировать с помощью методов интерфейса `EventTarget`. Все события, описываемые ниже, возбуждаются в объекте `WebSocket`. Ни одно из них не всплывает, и ни для одного из них не предусмотрено действий по умолчанию, которые можно было бы отменить. Отметьте, однако, что им передаются различные объекты событий.

`onclose`

Событие «close» генерируется после закрытия соединения (и свойство `readyState` получит значение `CLOSED`). Обработчику события передается объект `CloseEvent`, который определяет, было соединение закрыто без ошибок или нет.

`onerror`

Событие «error» генерируется, когда возникает сетевая ошибка или ошибка протокола веб-сокетов. Обработчику события передается обычный объект `Event`.

`onmessage`

Когда сервер отправляет данные через веб-сокет, объект `WebSocket` возбуждает событие «message» и передает обработчику объект `MessageEvent`, свойство `data` которого ссылается на принятое сообщение.

`onopen`

Конструктор `WebSocket()` возвращает управление еще до того, как будет установлено соединение с адресом *url*. Когда процедура установления соединения завершится и объект `WebSocket` будет готов к отправке и приему данных, будет возбуждено событие «open». Обработчику события передается обычный объект `Event`.

Window

окно, вкладка или фрейм веб-броузера **EventTarget**

Объект `Window` представляет окно, вкладку или фрейм в броузере. Он подробно описан в главе 14. В клиентском JavaScript-коде объект `Window` выступает в качестве «глобального» объекта, и все выражения вычисляются в контексте текущего объекта `Window`. Это значит, что для обращения к текущему окну не требуется использовать специальный синтаксис и свойства этого объекта можно использовать, как если бы они были глобальными переменными. Например, вместо *window*.`document` можно писать `document`. Аналогично можно вызывать методы текущего объекта окна, как если бы они были функциями, например `alert()` вместо *window*.`alert()`.

Некоторые свойства и методы этого объекта фактически позволяют определять и изменять некоторые параметры окна броузера. Другие включены в этот объект просто потому, что он является глобальным объектом. Помимо перечисленных здесь свойств и методов объект `Window` реализует все глобальные функции, определяемые базовым языком JavaScript. Подробности см. в справочной статье `Global` в третьей части книги.

Веб-броузеры возбуждают в окнах множество различных событий. Это означает, что объект `Window` определяет массу обработчиков событий и что объекты `Window` реализуют методы интерфейса `EventTarget`.

В объекте `Window` имеются свойства `window` и `self`, которые ссылаются на само окно. Они позволяют явно задать ссылку на окно.

Объект `Window` может содержать другие объекты `Window`, обычно в виде тегов `<iframe>`. Каждый объект `Window` является объектом, подобным массиву, содержащим вложенные объекты `Window`. Однако вместо непосредственного индексирования объекта `Window` на практике обычно используется свойство `frames`, ссылающееся на сам объект, как если бы это был объект, подобный массиву. Свойства `parent` и `top` объекта `Window` ссылаются непосредственно на родительское окно и на окно верхнего уровня.

Новые окна верхнего уровня создаются вызовом метода `Window.open()`. При вызове этого метода можно сохранить возвращаемое им значение в переменной и затем использовать эту переменную для ссылки на новое окно. Свойство `opener` нового окна будет ссылаться на окно, открывшее его.

Свойства

В дополнение к свойствам, перечисленным ниже, содержимое документа, отображаемого в окне, создает новые свойства. Как описывается в разделе 14.7, на элемент документа можно сослаться, использовав значение его атрибута `id` в качестве имени свойства окна (а поскольку его окно является глобальным объектом, его свойства являются глобальными переменными).

`readonly ApplicationCache` **`applicationCache`**

Ссылка на объект `ApplicationCache`. Кэшируемые и автономные веб-приложения могут использовать это свойство для управления обновлением своего кэша.

`readonly any` **`dialogArguments`**

В объектах `Window`, созданных методом `showModalDialog()`, это свойство хранит значение аргумента *`arguments`*, переданного методу `showModalDialog()`. В обычных объектах `Window` это свойство отсутствует. Подробнее см. в разделе 14.5.

`readonly Document` **`document`**

Ссылка на объект `Document`, который описывает документ, содержащийся в этом окне (подробности см. в справочной статье `Document`).

`readonly Event` **`event`** *[только в IE]*

В Internet Explorer это свойство ссылается на объект `Event`, содержащий сведения о самом последнем произошедшем в окне событии. В IE версии 8 и ниже объект события не всегда передается обработчикам событий, и поэтому иногда его приходится извлекать из этого свойства. Дополнительные сведения приводятся в главе 17.

`readonly Element` **`frameElement`**

Если данный объект `Window` находится внутри элемента `<iframe>`, это свойство будет ссылаться на представляющий его объект `IFrame`. В окнах верхнего уровня это свойство имеет значение `null`.

`readonly Window` **`frames`**

Подобно свойствам `self` и `window`, это свойство ссылается на сам объект `Window`. Все объекты `Window` являются объектами, подобными массивам, содержащими фреймы, имеющиеся в данном окне. Вместо ссылки `w[0]` на первый фрейм в окне `w` это свойство позволяет использовать более очевидную форму записи `w.frames[0]`.

`readonly History` **`history`**

Ссылка на объект `History` данного окна. См. `History`.

`readonly long` **`innerHeight`**
`readonly long` **`innerWidth`**

Высота и ширина в пикселах экранной области вывода окна. Эти свойства не поддерживаются в IE версии 8 и ниже. Порядок использования этих свойств демонстрируется в примере 15.9.

`readonly unsigned long` **`length`**

Количество фреймов, содержащихся в данном окне. См. описание свойства `frames`.

`readonly Storage` **`localStorage`**

Это свойство ссылается на объект `Storage`, предоставляющий доступ к хранилищу пар имя/значение на стороне клиента. Данные, сохраненные с помощью свойства `localStorage`, доступны любым документам с тем же происхождением и хранятся, пока не будут удалены пользователем или сценарием. См. также `sessionStorage` и раздел 20.1.

readonly Location **location**

Объект Location для данного окна. Этот объект определяет URL-адрес текущего загруженного документа. Запись нового URL-адреса в это свойство приводит к загрузке и выводу содержимого этого URL-адреса в броузере. См. Location.

string **name**

Имя окна. Имя может быть задано при создании окна методом open() или в виде значения атрибута name в теге <frame>. Имя окна может использоваться в качестве значения атрибута target в теге <a> или <form>. При таком применении атрибут target указывает, что документ, загружаемый по гиперссылке, или результаты отправки данных формы должны отображаться в указанном окне.

readonly Navigator **navigator**

Ссылка на объект Navigator, позволяющий получить информацию о версии и конфигурации веб-броузера. См. Navigator.

readonly Window **opener**

Доступная для чтения и записи ссылка на объект Window, в котором содержится сценарий, вызвавший метод open() для открытия в броузере окна верхнего уровня, или null в окнах, созданных иным способом. Это свойство действительно только для объектов Window, представляющих окна верхнего уровня, но не для объектов, представляющих фреймы. Свойство opener может использоваться во вновь созданном окне для доступа к свойствам и методам создавшего его окна.

readonly long **outerHeight**
readonly long **outerWidth**

Эти свойства определяют общую высоту и ширину окна броузера в пикселах. Эти размеры включают высоту и ширину строки меню, панелей инструментов, полос прокрутки, рамок окна и тому подобное. Эти свойства не поддерживаются в IE версии 8 и ниже.

readonly long **pageXOffset**
readonly long **pageYOffset**

Число пикселов, на которые текущий документ был прокручен вправо (pageXOffset) и вниз (pageYOffset). Эти свойства не поддерживаются в IE версии 8 и ниже. Порядок использования этих свойств и совместимый программный код, действующий в IE, демонстрируются в примере 15.8.

readonly Window **parent**

Объект Window, содержащий данное окно. Если данное окно является окном верхнего уровня, parent ссылается на само окно. Если данное окно является фреймом, свойство parent ссылается на окно или фрейм, в котором содержится данное окно.

string **returnValue**

Это свойство отсутствует в обычных окнах, но присутствует в объектах Window, созданных методом showModalDialog(), и по умолчанию содержит пустую строку. Когда окно диалога закрывается (см. описание метода close()), этому свойству присваивается значение, возвращаемое методом showModalDialog().

readonly Screen **screen**

Объект Screen, свойства которого содержат информацию об экране, включая число доступных пикселов и цветов. Подробности см. в справочной статье Screen.

readonly long **screenX**
readonly long **screenY**

Координаты верхнего левого угла окна на экране.

`readonly Window` **`self`**

Ссылка на само окно. Синоним свойства `window`.

`readonly Storage` **`sessionStorage`**

Это свойство ссылается на объект `Storage`, предоставляющий доступ к хранилищу пар имя/значение на стороне клиента. Данные, сохраненные с помощью свойства `sessionStorage`, доступны только документам в том же окне верхнего уровня или вкладке и хранятся только в течение сеанса работы с броузером. См. также `localStorage` и раздел 20.1.

`readonly Window` **`top`**

Окно верхнего уровня, содержащее данное окно. Если данное окно является окном верхнего уровня, свойство `top` содержит ссылку на само окно. Если данное окно представляет собой фрейм, свойство `top` ссылается на окно верхнего уровня, содержащее данный фрейм. Сравните со свойством `parent`.

`readonly object` **`URL`**

На момент написания этих строк данное свойство было ссылкой на объект, определяющий функции, которые были описаны в справочной статье `URL`. В будущем это свойство может превратиться в конструктор `URL()` и определять прикладной интерфейс для анализа URL-адресов и строк запроса в них.

`readonly Window` **`window`**

Свойство `window` идентично свойству `self` – оно содержит ссылку на данное окно. Поскольку в клиентских сценариях на языке JavaScript объект `Window` является глобальным объектом, данное свойство позволяет обращаться к глобальному объекту как к глобальной переменной `window`.

Конструкторы

Будучи глобальным объектом, объект `Window` должен определять все глобальные конструкторы, необходимые для клиентского окружения. Хотя здесь их перечень не приводится, следует понимать, что все глобальные конструкторы, описанные в этой части книги, являются свойствами объекта `Window`. Тот факт, что в клиентском JavaScript определены, к примеру, конструкторы `Image()` и `XMLHttpRequest()`, означает, что каждый объект `Window` имеет свойства с именами `Image` и `XMLHttpRequest`.

Методы

Объект `Window` определяет следующие методы, а также наследует все глобальные функции, определяемые в базовом языке JavaScript (см. справочную статью `Global` в третьей части книги).

`void` **`alert(`**`string` *`message`*`)`

Метод `alert()` показывает пользователю сообщение *`message`* в диалоговом окне. Диалоговое окно содержит кнопку OK, на которой пользователь может щелкнуть, чтобы закрыть окно. Обычно метод `alert()` выводит модальное диалоговое окно, и исполнение JavaScript-кода приостанавливается до тех пор, пока пользователь не закроет его.

`string` **`atob(`**`string` *`atob`*`)`

Эта вспомогательная функция принимает строку в формате base64 и декодирует ее в двоичную строку, где каждый символ представлен единственным байтом. Извлекать значения байтов из полученной строки можно с помощью ее метода `charCodeAt()`. См. также `btoa()`.

void **blur()**

Метод `blur()` убирает фокус ввода из окна верхнего уровня, соответствующего объекту `Window`. Точно не определено, какому окну передается фокус в результате вызова этого метода. В некоторых броузерах и/или на некоторых платформах данный метод может не оказывать никакого эффекта.

string **btoa**(string *btoa*)

Эта вспомогательная функция принимает двоичную строку (в которой каждый символ представлен единственным байтом) и возвращает ее в формате base64. Создать двоичную строку из произвольной последовательности байтов можно с помощью метода `String.fromCharCode()`. См. также `atob()`.

void **clearInterval**(long *handle*)

Метод `clearInterval()` останавливает периодическое выполнение программного кода, которое было начато вызовом метода `setInterval()`. В качестве аргумента *handle* должно передаваться значение, полученное при вызове метода `setInterval()`

void **clearTimeout**(long *handle*)

Метод `clearTimeout()` отменяет выполнение программного кода, отложенное методом `setTimeout()`. Аргумент *handle* – это значение, возвращаемое вызовом `setTimeout()` и идентифицирующее блок программного кода, отложенное исполнение которого отменяется.

void **close()**

Метод `close()` закрывает окно верхнего уровня, относительно которого он был вызван. Закрыты могут быть только те окна, которые были открыты JavaScript-сценарием.

boolean **confirm**(string *message*)

Выводит сообщение *message* в диалоговом окне, содержащем кнопки OK и Cancel (Отмена), с помощью которых пользователь должен ответить на вопрос. Если пользователь щелкнет на кнопке OK, метод `confirm()` вернет `true`. Если пользователь щелкнет на кнопке Cancel, метод `confirm()` вернет `false`.

void **focus()**

Передает фокус ввода окну верхнего уровня, соответствующему объекту `Window`. На большинстве платформ при получении фокуса окно верхнего уровня перемещается на вершину стека окон.

CSSStyleDeclaration **getComputedStyle**(Element *elt*, [string *pseudoElt*])

Элемент документа может получать информацию о стиле из встроенного атрибута `style` и из произвольного числа каскадных таблиц стилей. Прежде чем элемент будет отображен в окне, информация о стилях для этого элемента должна быть извлечена из каскадных таблиц стилей, а величины, выражаемые в относительных единицах (таких как проценты или «ems»), должны быть «вычислены» и преобразованы в абсолютные значения. Эти вычисленные значения иногда называют «используемыми» значениями.

Данный метод возвращает доступный только для чтения объект `CSSStyleDeclaration`, который представляет эти вычисленные CSS-стили, фактически используемые при отображении элементов. Все размеры в этих стилях выражены в пикселах.

Второй аргумент при вызове этого метода обычно опускается или в нем передается значение `null`, однако в нем можно также передать псевдоэлемент CSS «::before» или «::after», чтобы определить стили для содержимого.

Сравните метод `getComputedStyle()` со свойством `style` объекта `HTMLElement`, которое предоставляет доступ только к встроенным стилям элемента. Величины в этом свойстве могут измеряться в любых единицах, но оно ничего не сообщает о стилях из таблиц стилей, которые применяются к элементу.

Этот метод не реализован в IE версии 8 и ниже, но аналогичная функциональность доступна через нестандартное свойство `currentStyle`, имеющееся у каждого объекта `HTMLElement`.

`Window` **`open`**`([string` *`url`*`], [string` *`target`*`], [string` *`features`*`], [string` *`replace`*`])`

Метод `open()` загружает и отображает документ с адресом *url* в новом или существующем окне или вкладке. Аргумент *url* определяет URL-адрес документа, который требуется загрузить. Если он не указан, используется адрес «about:blank».

Аргумент *target* определяет имя окна, в которое требуется загрузить документ с адресом *url*. Если не указан, используется значение «_blank». Если аргумент *target* имеет значение «_blank», или если окно с указанным именем не найдено, для отображения документа с адресом *url* будет создано новое окно с указанным именем.

Аргумент *features* используется для определения позиции окна, размеров и других особенностей (таких как необходимость отображения строки меню, панелей инструментов и так далее). В современных броузерах, поддерживающих вкладки, этот аргумент обычно игнорируется и поэтому не описывается здесь.

При использовании метода `Window.open()` для загрузки нового документа в существующее окно методу можно передать аргумент *replace*, определяющий, должна ли для нового документа создаваться новая запись в истории просмотра окна или URL-адрес документа должен заменить текущую запись. Если аргумент *replace* равен `true`, то URL-адрес нового документа заменяет старую запись. Если этот аргумент равен `false` или не указан, то URL-адрес нового документа добавляется в историю просмотра окна в качестве новой записи. Этот аргумент обеспечивает методу функциональность, во многом схожую с функциональностью метода `Location.replace()`.

`void` **`postMessage`**`(any` *`message`*`, string` *`targetOrigin`*`, [MessagePort[]` *`ports`*`])`

Посылает данному окну копию сообщения *message* в порты *ports*, но только если документ, отображаемый в данном окне, имеет происхождение *targetOrigin*.

В аргументе *message* можно передать любой объект, который можно скопировать с применением алгоритма структурированного копирования (врезка «Структурированные копии» в главе 22). Аргумент *targetOrigin* должен быть абсолютным URL-адресом, содержащим протокол, имя хоста и порт, которые определяют требуемое происхождение. Если происхождение не имеет значения, в аргументе *targetOrigin* можно передать строку «*», а чтобы указать собственное происхождение сценария – строку «/». Вызов этого метода генерирует событие «message» в окне. См. также `MessageEvent` и раздел 22.3.

`void` **`print()`**

На вызов метода `print()` броузер реагирует так же, как если бы пользователь выбрал пункт меню или щелкнул на кнопке Print (Печать). Обычно после этого появляется диалоговое окно, позволяющее отменить операцию печати или выполнить дополнительную настройку.

`string` **`prompt`**`(string` *`message`*`, [string` *`default`*`])`

Метод `prompt()` выводит сообщение *message* в диалоговом окне, содержащем поле ввода и кнопки OK и Cancel, и блокирует работу сценария, пока пользователь не щелкнет на одной из кнопок.

Если пользователь щелкнет на кнопке Cancel, метод `prompt()` вернет `null`. Если пользователь щелкнет на кнопке OK, метод `prompt()` вернет значение, указанное в этот момент в поле ввода.

Аргумент *default* определяет начальное содержимое поля ввода.

void **scroll**(long *x*, long *y*)

Синоним метода scrollTo().

void **scrollBy**(long *x*, long *y*)

Прокручивает документ, отображаемый в окне, на относительную величину, заданную аргументами *x* и *y*.

void **scrollTo**(long *x*, long *y*)

Прокручивает документ, отображаемый в окне, так, чтобы точка с координатами *x* и *y* в документе оказалась в левом верхнем углу, если это возможно.

long **setInterval**(function *f*, unsigned long *interval*, any *args*...)

Метод setInterval() регистрирует функцию *f*, которая должна быть вызвана через *interval* миллисекунд и затем должна вызываться через каждые *interval* миллисекунд. Ключевое слово this внури функции *f* будет ссылаться на объект Window, а в аргументе *args* она получит все дополнительные аргументы, переданные методу setInterval().

Метод setInterval() возвращает число, которое позднее может быть передано методу Window.clearInterval() для прекращения периодического вызова функции *f*.

По историческим причинам в аргументе *f* можно передать не только функцию, но и строку с программным кодом на языке JavaScript. В этом случае каждые *interval* миллисекунд будет выполняться программный код, содержащийся в строке (как если бы он был заключен в тег <script>).

Если необходимо просто отложить выполнение программного кода и не требуется периодически запускать его, следует использовать метод setTimeout().

long **setTimeout**(function *f*, unsigned long *timeout*, any *args*...)

Метод setTimeout() напоминает метод setInterval(), но вызывает указанную функцию только один раз: он регистрирует функцию *f*, которая должна быть вызвана через *timeout* миллисекунд и возвращает число, которое позднее можно передать методу clearTimeout(), чтобы отменить вызов ожидающей функции. Когда истечет указанный интервал времени, функция *f* будет вызвана как метод объекта Window и ей будут переданы аргументы *args*. Если *f* – это строка с программным кодом, а не функция, она будет выполнена спустя *timeout* миллисекунд как сценарий в теге <script>.

any **showModalDialog**(string *url*, [any *arguments*])

Создает новый объект Window, сохраняет значение *arguments* в свойстве dialogArguments этого объекта, загружает в окно документ с адресом *url* и блокирует выполнение сценария, пока окно не будет закрыто. После закрытия окна метод возвращает значение свойства returnValue окна. Обсуждение и порядок использования метода можно найти в разделе 14.5 и в примере 14.4.

Обработчики событий

Большинство событий, возникающих в HTML-элементах, всплывают вверх по дереву документа до объекта Document и затем до объекта Window. По этой причине в объекте Window можно использовать любые свойства обработчиков событий, которые перечислены в справочной статье Element. И дополнительно можно использовать свойства обработчиков событий, перечисленные ниже. По историческим причинам каждое из свойств обработчиков событий, перечисленных ниже, можно также определить (в виде HTML-атрибутов или JavaScript-свойств) в элементе <body>.

Обработчик событий	Вызывается...
`onafterprint`	После вывода содержимого окна на печать
`onbeforeprint`	Перед выводом содержимого окна на печать
`onbeforeunload`	Перед тем как броузер покинет текущую страницу. Если возвращает строку или присваивает строку свойству `returnValue` объекта события, эта строка будет выведена в диалоге подтверждения. См. `BeforeUnloadEvent`.
`onblur`	Когда окно теряет фокус ввода
`onerror`	Когда возникает ошибка в JavaScript-сценарии. Это необычный обработчик события. См. раздел 14.6.
`onfocus`	Когда окно получает фокус ввода
`onhashchange`	Когда идентификатор фрагмента (см. `Location.hash`) документа изменяется в результате перемещения по истории посещений (см. `HashChangeEvent`)
`onLoad`	Когда документ и все внешние ресурсы будут загружены полностью
`onmessage`	Когда сценарий в другом окне отправит сообщение вызовом метода `postMessage()`. См. `MessageEvent`.
`onoffline`	Когда броузер потеряет соединение с Интернетом
`ononline`	Когда броузер восстановит соединение с Интернетом
`onpagehide`	Перед началом процедуры сохранения страницы в кэше и замещения ее другой страницей
`onpageshow`	Когда страница загружается впервые, событие «pageshow» возбуждается сразу после события «load», при этом свойство `persisted` объекта события имеет значение `false`. Однако когда ранее загруженная страница восстанавливается из кэша броузера, размещенного в памяти, событие «load» не возбуждается (поскольку страница в кэше считается уже загруженной), а событие «pageshow» возбуждается с объектом события, свойство `persisted` которого имеет значение `true`. См. `PageTransitionEvent`.
`onpopstate`	Когда броузер загружает новую страницу или восстанавливает состояние, сохраненное с помощью метода `History.pushState()` или `History.replaceState()`. См. `PopStateEvent`.
`onresize`	Когда пользователь изменяет размер окна броузера
`onscroll`	Когда пользователь прокручивает окно броузера
`onstorage`	Когда изменяется содержимое `localStorage` или `sessionStorage`. См. `StorageEvent`.
`onunload`	Броузер покинул страницу. Обратите внимание: если страница зарегистрирует обработчик события `onunload`, она не будет сохраняться в кэше. Чтобы обеспечить быстрый возврат к странице без повторной ее загрузки, следует использовать обработчик `onpagehide`.

Worker

фоновый поток выполнения **EventTarget**

Объект `Worker` представляет фоновый поток выполнения. Создать новый объект `Worker` можно с помощью конструктора `Worker()`, передав ему URL-адрес файла с программным кодом на языке JavaScript. Программный код в этом файле может использовать синхронные прикладные интерфейсы или выполнять продолжительные вычисления, не оказывая влияния на главный поток выполнения. Фоновые потоки работают в отдельном контексте выполнения (см. `WorkerGlobalScope`), и обмен данными с фоновым потоком выполнения возможен только через механизм асинхронных событий. Отправить данные фоновому потоку можно вызовом метода `postMessage()`, а получить – с помощью обработчика события «message».

Введение в фоновые потоки выполнения приводится в разделе 22.4.

Конструктор

`new` **`Worker`**`(string` *`scriptURL`*`)`

Создает новый объект `Worker` и запускает JavaScript-сценарий, находящийся по адресу *`scriptURL`*.

Методы

`void` **`postMessage`**`(any` *`message`*`, [MessagePort[]` *`ports`*`])`

Отправляет сообщение *`message`* фоновому потоку выполнения, котопый получит его в виде объекта `MessageEvent`, в обработчике `onmessage`. Аргумент *`message`* может быть простым значением, объектом или массивом, но не функцией. Допускается передавать такие объекты клиентского JavaScript, как `ArrayBuffer`, `File`, `Blob` и `ImageData`, но узлы, такие как `Document` и `Element`, передавать нельзя (подробности приводятся во врезке «Структурированные копии» в главе 22).

Необязательный аргумент *`ports`* позволяет указать один или более прямых каналов связи с объектом `Worker`. Например, если имеются два объекта `Worker`, можно обеспечить прямое взаимодействие между ними, передав их конструкторам концы соединения `MessageChannel`.

`void` **`terminate`**`()`

Останавливает фоновый поток выполнения и прерывает работу сценария в нем.

Обработчики событий

Поскольку фоновые потоки выполняются в окружении, отличном от окружения, создавшего их, они могут взаимодействовать с родительским потоком только посредством событий. Обработчики этих событий можно зарегистрировать с помощью свойств, перечисленных ниже, или с помощью методов интерфейса `EventTarget`.

`onerror`

Когда в сценарии, выполняемом в фоновом потоке, возбуждается исключение и это исключение не обрабатывается обработчиком `onerror` объекта `WorkerGlobalScope`, генерируется событие «error» в объекте `Worker`. Обработчику этого события передается объект `ErrorEvent`. Событие «error» не всплывает. Если данный фоновый поток выполнения запущен другим фоновым потоком, отмена события «error» предотвратит его передачу родительскому фоновому потоку. Если объект `Worker` создан в главном потоке выполнения, отмена события может предотвратить вывод сообщения в консоли JavaScript.

`onmessage`

Когда сценарий, выполняемый в фоновом потоке, вызовет свою глобальную функцию `postMessage()` (см. `WorkerGlobalScope`), в объекте `Worker` будет сгенерировано событие «message». Обработчику события будет передан объект `MessageEvent`, свойство `data` которого будет содержать копию значения, переданного сценарием из фонового потока выполнения методу `postMessage()`.

WorkerGlobalScope

EventTarget, Global

Фоновый поток, представляющий объект `Worker`, работает в среде выполнения, совершенно отличной от родительского потока, породившего его. Объект `WorkerGlobalScope` является глобальным объектом для фонового потока выполнения, поэтому получается, что данная справочная статья описывает среду выполнения «внутри» объекта `Worker`. Поскольку объект `WorkerGlobalScope` играет роль глобального объекта, он наследует свойства и методы глобального объекта базового языка JavaScript.

Свойства

В дополнение к свойствам, перечисленным ниже, объект `WorkerGlobalScope` определяет все глобальные свойства базового JavaScript, такие как `Math` и `JSON`.

`readonly WorkerLocation` **`location`**

Это свойство, подобно свойству `window.location`, является объектом `Location`: оно позволяет фоновому потоку проверить URL-адрес, откуда был загружен выполняемый в нем сценарий, и включает в себя свойства, возвращающие отдельные части URL.

`readonly WorkerNavigator` **`navigator`**

Это свойство, подобно свойству `window.navigator`, является объектом `Navigator`: оно определяет свойства, позволяющие фоновому потоку определить тип броузера, в котором он выполняется, и состояние подключения к сети.

`readonly WorkerGlobalScope` **`self`**

Это свойство ссылается на сам глобальный объект `WorkerGlobalScope`. Оно похоже на свойство `window` объекта `Window` в главном потоке выполнения.

Методы

В дополнение к методам, перечисленным ниже, объект `WorkerGlobalScope` определяет все глобальные функции базового JavaScript, такие как `isNaN()` и `eval()`.

`void` **`clearInterval(`**`long` *`handle`***`)`**

В точности соответствует одноименному методу объекта `Window`.

`void` **`clearTimeout(`**`long` *`handle`***`)`**

В точности соответствует одноименному методу объекта `Window`.

`void` **`close()`**

Переводит поток выполнения в особое состояние «завершения». Оказавшись в этом состоянии он больше не будет возбуждать события. Сценарий продолжит работу до момента возврата в цикл событий фонового потока выполнения, где тут же будет остановлен.

`void` **`importScripts(`**`string` *`urls...`***`)`**

Для каждого из аргументов *`urls`* этот метод разрешает URL-адрес относительно свойства `location`, затем загружает содержимое URL-адреса и выполняет его, как программный код на языке JavaScript. Обратите внимание, что это синхронный метод. Он загружает и выполняет файлы по очереди и не возвращает управление, пока не выполнит все сценарии. (Однако, если какой-то сценарий возбудит исключение, это исключение начнет распространение и помешает загрузке и выполнению следующих за ним сценариев.)

`void` **`postMessage`**`(any` *`message`*`, [MessagePort[]` *`ports`*`])`

Отправляет сообщение *message* (и массив портов, если указан) потоку выполнения, породившему данный фоновый поток. Вызов этого метода генерирует событие «message» в объекте `Worker` в родительском потоке выполнения, обработчику которого передается объект `MessageEvent` со свойством `data`, содержащим копию аргумента *message*. Обратите внимание, что в фоновом потоке выполнения метод `postMessage()` является глобальной функцией.

`long` **`setInterval`**`(any` *`handler`*`, [any` *`timeout`*`], any` *`args`*`...)`

В точности соответствует одноименному методу объекта `Window`.

`long` **`setTimeout`**`(any` *`handler`*`, [any` *`timeout`*`], any` *`args`*`...)`

В точности соответствует одноименному методу объекта `Window`.

Конструкторы

Объект `WorkerGlobalScope` содержит все конструкторы базового JavaScript, такие как `Array()`, `Date()` и `RegExp()`. Он также определяет некоторые наиболее важные конструкторы клиентского JavaScript, позволяющие создавать объекты `XMLHttpRequest`, `FileReaderSync` и даже сам объект `Worker`.

Обработчики событий

Обработчики событий для фонового потока выполнения можно зарегистрировать, установив следующие глобальные свойства или воспользовавшись методами интерфейса `EventTarget`, реализованными в объекте `WorkerGlobalScope`.

`onerror`

Это необычный обработчик события: это свойство больше похоже на свойство `onerror` объекта `Window`, чем на свойство `onerror` объекта `Worker`. Когда в фоновом потоке выполнения появляется необработанное исключение, будет вызвана эта функция, если она определена, с тремя строковыми аргументами, определяющими сообщение об ошибке, URL-адрес сценария и номер строки в сценарии. Если функция вернет `false`, исключение будет считаться обработанным и прекратит дальнейшее распространение. В противном случае, если это свойство не установлено или обработчик не вернул `false`, исключение продолжит распространение и вызовет событие «error» в объекте Worker в родительском потоке выполнение.

`onmessage`

Когда родительский поток выполнения вызывает метод `postMessage()` объекта `Worker`, представляющего данный фоновый поток выполнения, в данном объекте `WorkerGlobalScope` генерируется событие «message». Обработчику этого события будет передан объект `MessageEvent`, свойство `data` которого хранит копию аргумента *message*, переданного родительским потоком выполнения.

WorkerLocation

URL-адрес главного сценария в фоновом потоке выполнения

Объект `WorkerLocation`, на который ссылается свойство `location` объекта `WorkerGlobalScope`, похож на объект `Location`, на который ссылается свойство `location` объекта `Window`: он представляет URL-адрес главного сценария в фоновом потоке выполнения и определяет свойства, представляющие различные части этого URL-адреса.

Объект `Worker` отличается от объекта `Window` тем, что он не может перейти по другому адресу или перезагрузить сценарий, поэтому свойства объекта `WorkerLocation` доступны только для чтения, и в этом объекте отсутствуют методы интерфейса `Location`.

В отличие от обычного объекта `Location`, объект `WorkerLocation` не преобразуется в строку автоматически. В фоновом потоке выполнения нельзя просто обратиться к имени `location` там, где подразумевается `location.href`.

Свойства

Следующие свойства имеют то же назначение, что и одноименные свойства объекта `Location`.

`readonly string` **`hash`**

Часть URL-адреса – идентификатор фрагмента, включающий начальный символ решетки.

`readonly string` **`host`**

Часть URL-адреса – имя хоста и порт.

`readonly string` **`hostname`**

Часть URL-адреса – имя хоста.

`readonly string` **`href`**

Полный текст URL-адреса, переданный конструктору `Worker()`. Это единственное значение, которое фоновый поток выполнения получает непосредственно от родительского потока: все остальные значения передаются косвенно – посредством событий «message».

`readonly string` **`pathname`**

Часть URL-адреса – путь.

`readonly string` **`port`**

Часть URL-адреса – порт.

`readonly string` **`protocol`**

Часть URL-адреса – протокол.

`readonly string` **`search`**

Часть URL-адреса – строка поиска или запроса, включая начальный знак вопроса.

WorkerNavigator

информация о броузере для фонового потока выполнения

Свойство `navigator` объекта `WorkerGlobalScope` ссылается на объект `WorkerNavigator`, который является упрощенной версией объекта `Navigator` окна.

Свойства

Следующие свойства имеют то же назначение, что и одноименные свойства объекта `Navigator`.

`readonly string` **`appName`**

См. описание свойства `appName` объекта `Navigator`.

`readonly string` **`appVersion`**

См. описание свойства `appVersions` объекта `Navigator`.

`readonly boolean` **`onLine`**

Имеет значение `true`, если броузер подключен к сети, и `false` – в противном случае.

`readonly string` **`platform`**

Строка, идентифицирующая операционную систему и/или аппаратную платформу, на которой выполняется броузер.

`readonly string` **`userAgent`**

Значение, используемое броузером для заголовка `user-agent` в HTTP-запросах.

XMLHttpRequest

позволяет выполнять HTTP-запросы и получать ответы **EventTarget**

Объект `XMLHttpRequest` позволяет из клиентских JavaScript-сценариев запускать HTTP-запросы и получать от веб-сервера ответы (которые не обязательно должны быть в формате XML). Объект `XMLHttpRequest` подробно рассматривается в главе 18, там же можно найти множество примеров применения этого объекта.

Создать объект `XMLHttpRequest` можно с помощью конструктора `XMLHttpRequest()` (сведения о том, как создавать объекты `XMLHttpRequest` в IE6, приводятся во врезке в разделе 18.1) и затем использовать его следующим образом:

1. Вызывается метод `open()`, с помощью которого определяются URL-адрес и метод передачи запроса (обычно «GET» или «POST»).
2. В свойство `onreadystatechange` записывается ссылка на функцию, которая будет вызываться в процессе выполнения запроса.
3. Вызывается метод `setRequestHeader()`, если необходимо указать дополнительные параметры запроса.
4. Вызовом метода `send()` выполняется отправка запроса веб-серверу. Если был выбран метод отправки POST, этому методу можно дополнительно передать тело запроса. В процессе выполнения запроса будет вызываться функция-обработчик события `onreadystatechange`. Когда свойство `readyState` получит значение 4, выполнение запроса завершится.
5. После того как свойство `readyState` достигнет значения 4, можно проверить код состояния в свойстве `status`, чтобы убедиться, что запрос завершился успехом. В этом случае методом `getResponseHeader()` или `getResponseHeaders()` следует извлечь значения из заголовка ответа и с помощью свойства `responseText` или `responseXML` получить тело ответа.

Объект `XMLHttpRequest` определяет относительно высокоуровневый прикладной интерфейс к протоколу HTTP. Он учитывает такие особенности, как обработка переадресации, управление cookies и обслуживание междоменных запросов с заголовками CORS.

Возможности объекта `XMLHttpRequest`, описанные выше, прекрасно поддерживаются всеми современными броузерами. На момент написания этих строк велись работы над стандартом «XMLHttpRequest Level 2», и производители броузеров уже приступили к его реализации. Свойства, методы и обработчики событий, перечисленные ниже, включают особенности, введенные спецификацией «XMLHttpRequest Level 2», которые могут быть реализованы не во всех броузерах. Эти новые особенности помечены строкой «XHR2».

Конструктор

`new` **`XMLHttpRequest()`**

Этот конструктор, не имеющий аргументов, возвращает новый объект `XMLHttpRequest`.

Константы

Следующие константы определяют возможные значения свойства `readyState`. До появления спецификации XHR2 эти константы не были стандартизованы и в большинстве программ вместо символических значений использовались целочисленные литералы.

unsigned short **UNSENT = 0**

Начальное состояние. Объект XMLHttpRequest только что создан или сброшен в исходное состояние вызовом метода abort().

unsigned short **OPENED = 1**

Метод open() уже вызван, но обращения к методу send() еще не было. Запрос еще не отправлен.

unsigned short **HEADERS_RECEIVED = 2**

Вызван метод send() и приняты заголовки ответа, но тело ответа еще не принято.

unsigned short **LOADING = 3**

Начат прием тела ответа, но прием еще не завершился.

unsigned short **DONE = 4**

HTTP-ответ принят полностью или прием был остановлен из-за ошибки.

Свойства

readonly unsigned short **readyState**

Состояние HTTP-запроса. В момент создания объекта XMLHttpRequest это свойство приобретает значение 0, а к моменту получения полного HTTP-ответа это значение возрастает до 4. Возможные значения свойства определяют константы, перечисленные выше.

Значение свойства readyState может уменьшаться, только если в процессе выполнения запроса был вызван метод abort() или open().

Теоретически при каждом изменении значения этого свойства должен вызываться обработчик события onreadystatechange. Однако на практике событие гарантированно возникает, только когда свойство readyState получает значение 4. (События «progress», введенные спецификацией XHR2, обеспечивают более надежный способ слежения за ходом выполнения запроса.)

readonly any **response**

В спецификации XHR2 это свойство хранит ответ сервера. Тип свойства зависит от значения свойства responseType. Если responseType содержит пустую строку или строку «text», данное свойство содержит тело ответа в виде строки. Если responseType содержит строку «document», значением данного свойства будет объект Document, полученный в результате разбора XML- или HTML-документа в теле ответа. Если responseType содержит строку «arraybuffer», значением данного свойства будет объект ArrayBuffer, представляющий двоичные данные в теле ответа. А если responseType содержит строку «blob», значением данного свойства будет объект Blob, представляющий двоичные данные в теле ответа.

readonly string responseText

Если значение свойства readyState меньше 3, данное свойство будет содержать пустую строку. Если значение свойства readyState равно 3, данное свойство возвращает часть ответа, которая была принята к текущему моменту. Если значение свойства readyState равно 4, это свойство содержит полное тело ответа.

Если в ответе имеется заголовок, определяющий кодировку символов в теле ответа, используется эта кодировка, в противном случае предполагается кодировка UTF-8.

string **responseType**

В спецификации XHR2 это свойство определяет тип ответа и тип свойства response. Допустимыми значениями являются «text», «document», «arraybuffer» и «blob».

Значением по умолчанию является пустая строка, которая также является синонимом значения «text». Если установить это свойство вручную, последующие попытки обратиться к свойствам `responseText` и `responseXML` будут возбуждать исключения и для получения ответа сервера необходимо будет использовать свойство `response`, предусмотренное спецификацией XHR2.

`readonly Document` **`responseXML`**

Ответ на запрос, который интерпретируется как XML- или HTML-документ и возвращается в виде объекта `Document`. Это свойство будет иметь значение `null`, если тело ответа еще не получено или оно не является допустимым XML или HTML-документом.

`readonly unsigned short` **`status`**

HTTP-код состояния, полученный от сервера, такой как 200 – в случае успеха, 404 – в случае ошибки отсутствия документа или 0 – если сервер еще не прислал код состояния.

`readonly string` **`statusText`**

Это свойство содержит текст, соответствующий HTTP-коду состояния в ответе. То есть, когда свойство `status` имеет значение 200, это свойство содержит строку «OK», а когда 404 – строку «Not Found». Это свойство содержит пустую строку, если сервер еще не прислал код состояния.

`unsigned long` **`timeout`**

Свойство, введенное спецификацией XHR2, определяющее предельное время ожидания ответа в миллисекундах. Если выполнение HTTP-запроса займет больше времени, чем указано в данном свойстве, он будет прерван и будет сгенерировано событие «timeout». Это свойство можно установить только после вызова метода `open()` и перед вызовом метода `send()`.

`readonly XMLHttpRequestUpload` **`upload`**

Свойство, введенное спецификацией XHR2, ссылающееся на объект `XMLHttpRequestUpload`, который определяет набор свойств регистрации обработчиков событий для слежения за процессом выгрузки тела HTTP-запроса.

`boolean` **`withCredentials`**

Свойство, введенное спецификацией XHR2, определяющее необходимость аутентификации при выполнении междоменного CORS-запроса и необходимость обработки заголовков `cookie` в CORS-ответах. По умолчанию имеет значение `false`.

Методы

`void` **`abort()`**

Возвращает объект `XMLHttpRequest` в исходное состояние, соответствующее значению 0 в свойстве `readyState`, и отменяет любые запланированные сетевые взаимодействия. Этот метод может потребоваться, например, если запрос выполняется слишком долго и надобность в получении ответа уже отпала.

`string` **`getAllResponseHeaders()`**

Возвращает все HTTP-заголовки ответа (с отфильтрованными заголовками `cookie` и `CORS`), полученные от сервера, или `null`, если заголовки еще не были получены. Заголовки `cookie` и `CORS` отфильтровываются и не могут быть получены. Заголовки возвращаются в виде единственной строки и отделяются друг от друга комбинацией символов `\r\n`.

`string` **`getResponseHeader`**`(string` *header*`)`

Возвращает значение указанного заголовка *header* в HTTP-ответе или `null`, если заголовки вообще не были получены или если ответ не содержит требуемого заголовка *header*. Заголовки `cookie` и `CORS` отфильтровываются, и их нет смысла запрашивать. Если было принято несколько заголовков с указанным именем, значения этих заголовков объединяются в одну строку через запятую и пробел.

`void` **`open`**`(string` *method*, `string` *url*, `[boolean` *async*, `string` *user*, `string` *pass*`])`

Этот метод инициализирует объект `XMLHttpRequest` и сохраняет свои аргументы для последующего использования методом `send()`.

Аргумент *method* определяет HTTP-метод, используемый для отправки запроса. Среди наиболее устоявшихся методов можно назвать GET, POST и HEAD. Реализации могут также поддерживать методы CONNECT, DELETE, OPTIONS, PUT, TRACE и TRACK.

Аргумент *url* определяет URL-адрес, который является предметом запроса. Разрешение относительных URL-адресов производится обычным образом с использованием URL-адреса документа со сценарием. Политика общего происхождения (см. раздел 13.6.2) требует, чтобы данный URL-адрес содержал те же имя хоста и номер порта, что и документ со сценарием, выполняющим запрос. Объект XHR2 позволяет выполнять междоменные запросы к серверам, поддерживающим заголовки `CORS`.

Если аргумент *async* указан и имеет значение `false`, запрос будет выполняться в синхронном режиме, и последующий вызов `send()` заблокирует работу сценария, пока ответ не будет получен полностью. Синхронные запросы рекомендуется использовать только в фоновых потоках выполнения.

Необязательные аргументы *user* и *pass* определяют имя пользователя и пароль для HTTP-запроса.

`void` **`overrideMimeType`**`(string` *mime*`)`

Этот метод позволяет указать, что ответ сервера должен интерпретироваться в соответствии с указанным MIME-типом *mime* (и параметром `charset`, если он указан в определении типа *mime*), без учета значения заголовка `Content-Type` в ответе.

`void` **`send`**`(any` *body*`)`

Инициирует выполнение HTTP-запроса. Если перед этим не вызывался метод `open()` или, обобщенно, если значение свойства `readyState` не равно 1, метод `send()` возбуждает исключение. В противном случае он начинает выполнение HTTP-запроса, который состоит из:

- HTTP-метода, URL-адреса и информации об авторизации (если необходимо), определенных предшествующим вызовом метода `open()`;
- заголовков запроса, если они были определены предшествующим вызовом метода `setRequestHeader()`;
- значения аргумента *body*, переданного данному методу. Аргумент *body* может быть строкой, объектом `Document`, образующим тело запроса; он может быть опущен или иметь значение `null`, если запрос не имеет тела (например, GET-запросы вообще не имеют тела). Согласно спецификации XHR2 телом запроса также могут быть объекты `ArrayBuffer`, `Blob` и `FormData`.

Если в предшествующем вызове метода `open()` аргумент *async* имел значение `false`, данный метод блокируется и не возвращает управление, пока значение свойства `readyState` не станет равно 4 и ответ сервера не будет получен полностью. В противном случае метод `send()` немедленно возвращает управление, а ответ сервера обрабатывается асинхронно, с помощью обработчиков событий.

void **setRequestHeader**(string *name*, string *value*)

Определяет HTTP-заголовок с именем *name* и значением *value*, который должен быть включен в запрос, передаваемый последующим вызовом метода `send()`. Этот метод может вызываться, только когда свойство `readyState` имеет значение 1, т. е. после вызова метода `open()`, но перед вызовом метода `send()`.

Если заголовок с именем *name* уже был определен, новым значением заголовка станет прежнее значение заголовка плюс запятая с пробелом и новое значение *value*, переданное методу.

Если методу `open()` была передана информация об авторизации, объект `XMLHttpRequest` автоматически добавит заголовок `Authorization`. Однако этот заголовок может быть также добавлен методом `setRequestHeader()`.

Объект `XMLHttpRequest` автоматически устанавливает заголовки «Content-Length», «Date», «Referer» и «User-Agent» и не позволяет изменять их значения. Существует еще несколько заголовков, включая заголовки, имеющие отношение к cookies, которые нельзя установить с помощью этого метода. Полный их список приводится в разделе 18.1.

Обработчики событий

Оригинальный объект `XMLHttpRequest` определяет только одно свойство регистрации обработчика событий: `onreadystatechange`. Спецификация XHR2 дополняет этот список множеством обработчиков событий хода выполнения запроса, которые намного проще в использовании. Зарегистрировать обработчики можно с помощью свойств, перечисленных ниже, или с помощью методов интерфейса `EventTarget`. События, возникающие в объекте `XMLHttpRequest`, всегда доставляются самому объекту `XMLHttpRequest`. Они не всплывают и не предусматривают действий по умолчанию, которые можно было бы отменить. Обработчикам событий «readystatechange» передается объект `Event`, а обработчикам остальных событий – объект `ProgressEvent`.

См. также описание свойства `upload` и `XMLHttpRequestUpload`, где приводится список событий, которые можно использовать для слежения за ходом выгрузки тела HTTP-запроса.

onabort

Вызывается при прерывании запроса.

onerror

Вызывается в случае завершения запроса по ошибке. Обратите внимание, что HTTP-коды состояния, такие как 404, не считаются ошибкой, поскольку сам ответ получен успешно. Однако это событие может породить отрицательный ответ сервера DNS или бесконечный цикл переадресаций.

onload

Вызывается при успешном выполнении запроса.

onloadend

Вызывается в случае успешного или неудачного завершения запроса, после событий «load», «abort», «error» и «timeout».

onloadstart

Вызывается с началом выполнения запроса.

onprogress

Вызывается периодически (примерно раз в 50 миллисекунд) в ходе загрузки тела ответа.

`onreadystatechange`

Вызывается при изменении значения свойства `readyState`. Наиболее важен для обработки ситуации завершения запроса.

`ontimeout`

Вызывается, если истекло время ожидания, определяемое свойством `timeout`, а ответ так и не был принят.

XMLHttpRequestUpload

EventTarget

Объект `XMLHttpRequestUpload` определяет множество свойств регистрации обработчиков событий для слежения за ходом выгрузки тела HTTP-запроса. В броузерах, реализующих положения спецификации «XMLHttpRequest Level 2», каждый объект `XMLHttpRequest` имеет свойство `upload`, ссылающееся на объект этого типа. Чтобы реализовать слежение за ходом выполнения операции выгрузки, достаточно просто установить соответствующие обработчики событий с помощью следующих свойств или методов интерфейса `EventTarget`. Обратите внимание, что перечисленные ниже свойства регистрации обработчиков событий для слежения за процессом выгрузки в точности соответствуют свойствам регистрации обработчиков событий для слежения за процессом загрузки, которые определяются самим объектом `XMLHttpRequest`, за исключением свойства `onreadystatechange`.

Обработчики событий

`onabort`

Вызывается при прерывании выгрузки.

`onerror`

Вызывается, когда в процессе выгрузки возникает сетевая ошибка.

`onload`

Вызывается в случае успешного завершения выгрузки

`onloadend`

Вызывается в случае успешного или неудачного завершения выгрузки. Событие «loadend» всегда следует за событиями «load», «abort», «error» и «timeout».

`onloadstart`

Вызывается с началом выгрузки.

`onprogress`

Вызывается периодически (примерно раз в 50 миллисекунд) в ходе выгрузки.

`ontimeout`

Вызывается, если истекло время ожидания, определяемое свойством `timeout` объекта `XMLHttpRequest`.

Алфавитный указатель

Символы

- $ (знак доллара)
  - $$(), метод объекта ConsoleCommandLine, 908
  - $(), метод объекта ConsoleCommandLine, 907
  - $(), функция, 32, 558, 559
    - использование вместо querySelectorAll(), 564
    - поиск элементов по идентификаторам, 381
  - $0 и $1, свойства объекта ConsoleCommandLine, 907
  - обозначение конца строки в регулярных выражениях, 277, 284
- & (амперсанд)
  - && (логическое И), оператор, 62, 85, 98
  - &= (поразрядное И и присваивание), оператор, 85, 101
  - поразрядное И, оператор, 85, 92
- | (вертикальная черта)
  - || (логическое ИЛИ), оператор, 85, 99
  - |= (поразрядное ИЛИ и присваивание), оператор, 85, 101
  - поразрядное ИЛИ, оператор, 85, 92
  - разделитель альтернатив в регулярных выражениях, 281, 283
- ! (восклицательный знак)
  - логическое НЕ, оператор, 69, 85, 100
  - !== (не идентично), оператор, 85, 93
    - проверка свойств на неравенство значению undefined, 148
  - != (не равно), оператор, 85, 93
- - (дефис)
  - диапазон в классах символов регулярных выражений, 279
- , (запятая) оператор, 108
- * (звездочка)
  - квантификатор в регулярных выражениях, 280
  - символ шаблона в расширении E4X, 312
  - *= (умножение и присваивание), оператор, 85, 101
  - умножение, оператор, 53, 85
- ? (знак вопроса)
  - квантификатор в регулярных выражениях, 280
  - ?= отрицательное условие на последующие символы в регулярных выражениях, 284
  - ?! положительное условие на последующие символы в регулярных выражениях, 284
  - ?: (условный) оператор, 85, 105
- ^ (знак вставки)
  - в регулярных выражениях, 284
  - отрицание в классах символов в регулярных выражениях, 278
  - ^= (поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ и присваивание), оператор, 85, 101
  - поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ, оператор, 85, 92
- - (знак минус)
  - -= (вычитание и присваивание), оператор, 101
  - -- (декремент), оператор, 84, 91
    - побочные эффекты, 87, 110
  - вычитание, оператор, 53, 85
  - унарный минус, оператор, 84, 90, 91
- + (знак плюс)
  - ++ (инкремент), оператор, 84, 90, 91
    - побочные эффекты, 87, 110
  - квантификатор в регулярных выражениях, 280
  - оператор конкатенации строк, 85
  - оператор сложения, 53, 85
    - типы данных операндов, 86
  - оператор сложения и конкатенации строк, 89
  - += (сложение и присваивание), оператор, 85, 101
    - добавление текста в свойство innerHTML, 409
  - унарный плюс, оператор, 84, 90
- % (знак процента)
  - %= (деление по модулю и присваивание), оператор, 85, 101
  - оператор деления по модулю, 53, 85
- = (знак равно)
  - === (идентично), оператор, 85, 93
  - оператор присваивания, 85
    - левостороннее выражение, 86
    - побочные эффекты, 87
  - == (равно), оператор, 85, 93
    - и значение NaN, 54
- // и /* */ в комментариях, 43
- " (кавычки, двойные), строковые литералы, 56
  - в регулярных выражениях, 282
  - в таблицах стилей или в атрибуте style, 463
- ' (кавычки, одинарные), строковые литералы, 56
  - в регулярных выражениях, 282
- [] (квадратные скобки)
  - в выражениях доступа к свойствам, 82, 191
  - генераторы массивов, 307
  - доступ к отдельным символам в строке, 184
  - инициализаторы массивов, 81
  - классы символов в регулярных выражениях, 278
  - получение и изменение значений свойств, 142, 143
  - создание массивов, 165
  - чтение и запись элементов массивов, 166

  чтение и запись элементов многомерных массивов, 171
@ (коммерческое «at»)
  @if, @else и @end, ключевые слова в условных комментариях, 360
  в именах атрибутов, 312
() (круглые скобки)
  в выражениях вызова функций и методов, 83
  в выражениях-генераторах, 308
  в выражениях создания объектов, 84
  в определениях функций, 82, 186
  группировка в регулярных выражениях, 281, 283
  окружающие выражения в условных инструкциях if, 115
\\ (обратный слэш)
  \\n (ссылки в регулярных выражениях, 283
  создание многострочных текстовых литералов, 57
  управляющие последовательности в регулярных выражениях, 277
  управляющие последовательности в строковых литералах, 58
_ (символ подчеркивания)
  в именах функций, 186
/ (слэш)
  /= (деление и присваивание), оператор, 85, 101
  /* */, комментарии в URL-адресах вида JavaScript:, 342
  литералы регулярных выражений, 276
  оператор деления, 53, 85
~ (тильда), оператор поразрядное НЕ, 84, 92
. (точка)
  .. (вложенный элемент), оператор, 312
  в выражениях обращения к свойствам, 82
  в регулярных выражениях классы символов, 279
  оператор точка
    доступ к свойствам при вызове методов, 190
    получение и изменение значений свойств, 142
; (точка с запятой)
  завершение инструкций, 109
    необязательное, 46
  и блоки инструкций, 111
  разделение пар имя/значение в свойствах стиля, 445
<> (угловые скобки)
  >= (больше или равно), оператор, 85, 95
  > (больше чем), оператор, 85, 95
  <= (меньше или равно), оператор, 85, 95
  < (меньше чем), оператор, 85, 95
  <<= (сдвиг влево и присваивание), оператор, 85, 101
  << (сдвиг влево), оператор, 85, 92
  >>>= (сдвиг вправо с заполнением нулями и присваивание), оператор, 85, 101
  >>> (сдвиг вправо с заполнением нулями), оператор, 85, 93
  >>= (сдвиг вправо с сохранением знака и присваивание), оператор, 85, 86
  >> (сдвиг вправо с сохранением знака), оператор, 85, 93
{} (фигурные скобки)
  блок инструкций, 110
  в определениях функций, 186
  инициализаторы объектов, 81
  количество повторений в регулярных выражениях, 280
  отсутствие в краткой форме записи функций, 309

A

<a>, элемент
  href, атрибут, 342, 396
  наличие атрибута name вместо атрибута href, 396
abort(), метод
  FileReader, объект, 947
  XMLHttpRequest, объект, 541, 544, 1036
about:blank, URL-адрес, 383
abs(), функция, объект Math, 823
accept, свойство объекта Input, 964
acceptCharset, свойство объекта Form, 950
accuracy, свойство объекта Geocoordinates, 955
acos(), функция, объект Math, 823
action, свойство объекта Form, 431, 950
activeElement, свойство объекта Document, 915
add(), метод
  DOMTokenList, объект, 470, 924
  HTMLOptionsCollection, объект, 960
  jQuery, объект, 619
  Select, элемент, 1009
addClass(), метод объекта jQuery, 567
addColorStop(), метод объекта CanvasGradient, 690, 890
addElement(), метод объекта DataTransfer, 509, 912
addEventListener(), метод, 348, 489
  EventTarget, объект, 944
  WorkerGlobalScope, объект, 724
  Worker, объект, 722
  несовместимость с методом attachEvent(), 356
  отсутствует в Internet Explorer, 353
  перехватывающие обработчики событий, 496
  регистрация обработчиков событий, 491
adoptNode(), метод объекта Document, 917
after(), метод объекта jQuery, 572
Ajax, архитектура
  в версии jQuery 1.5, 603
  и XML, 527
  определение, 524
  транспорты, 525
  утилиты в библиотеке jQuery, 595, 597
    ajax(), функция, 601
    get() и post(), функции, 600
    getJSON(), функция, 598
    getScript(), функция, 598
    load(), метод, 595
    коды состояния, 596
    передача данных, 599
    события Ajax, 608
    типы данных, 601
  функции поддержки в библиотеке jQuery, 980
ajax(), функция, 601
  редко используемые параметры и обработчики, 607
  функции обратного вызова, 605
  часто используемые параметры, 603
alert(), метод
  Window, объект, 334, 377, 1025
  бесконечный цикл, 367
altitude, свойство объекта Geocoordinates, 955
altitudeAccuracy, свойство объекта Geocoordinates, 955

altKey, свойство, 500, 519
  Event, объект, 937
  событий мыши, 483
andSelf(), метод объекта jQuery, 622
animate(), метод
  объект, определяющий изменяемые атрибуты, 590
  объект с параметрами анимационного эффекта, 591
  объекта jQuery, 586, 587
  реализация собственных анимационных эффектов, 589
append(), метод
  BlobBuilder, объект, 736, 887
  FormData, объект, 954
  jQuery, объект, 572
appendChild(), метод объекта Node, 413, 999
appendData(), метод
  Comment, узел, 905
  Text, объект, 1015
appendTo(), метод объекта jQuery, 572
<applet>, элемент, 571
ApplicationCache, объект, 883
  status, свойство, 649
  swapCache(), метод, 649, 883
  update(), метод, 649, 884
  константы, значения для свойства status, 883
  методы, 883
  обработчики событий, 884
  свойства, 883
applicationCache, свойство объекта Window, 646, 1023
apply(), метод объекта Function, 811
apply() и call(), методы объекта Function, 193, 210
  ограничение на использование в безопасных подмножествах, 293
appName, свойство
  Navigator, объект, 375, 996
  WorkerNavigator, объект, 1033
appVersion, свойство
  Navigator, объект, 375, 996
  WorkerNavigator, объект, 1033
arc(), метод объекта CanvasRenderingContext2D, 685, 686, 896
arcTo(), метод объекта CanvasRenderingContext2D, 685, 897
Arguments, объект, 194, 761
  callee и caller, свойства, 196
  callee, свойство, 761, 762
  length, свойство, 209, 761, 762
  ограничение на использование в безопасных подмножествах, 293
arguments, свойство объекта Function, 811
arguments[], массив, 761
ArrayBuffer, объект, 730, 885
  readAsArrayBuffer(), метод объекта FileReader, 740, 741
  порядок следования байтов, 731
  свойства, 885
ArrayBufferView, объект, 885
Array, объект
  isArray(), метод, 181
  isArray(), функция, 181
  sort(), метод, 247
Array(), конструктор, 165
asin(), функция, объект Math, 823
assert(), метод объекта Console, 905
assign(), метод объекта Location, 372, 987
async, свойство объекта Script, 1007
atan(), функция, объект Math, 824
atan2(), функция, объект Math, 824
atob(), метод объекта Window, 1025
attachEvent(), метод, 349, 489, 945
  значение ссылки this в обработчиках событий, 494
  несовместимость с методом addEventListener(), 356
  регистрация обработчиков событий в IE, 492
Attr, объект, 408, 885
attr(), метод объекта jQuery, 565, 576
attributes, свойство объекта Element, 408, 925
<audio>, элемент, 658
  controls, атрибут, 658
  события, 486
Audio, объект, 886
audio, свойство объекта Video, 1019
Audio(), конструктор, 659
Authorization, заголовки, 530
autocomplete, свойство
  Form, объект, 950
  Input, объект, 964
autofocus, свойство объекта FormControl, 952
autoplay, свойство объекта MediaElement, 661, 988
availHeight, свойство объекта Screen, 377, 1006
availWidth, свойство объекта Screen, 377, 1006

B

\\b (граница слова) в регулярных выражениях, 284
\\b (забой) в регулярных выражениях, 279
\\B (не слово) метасимвол, 284
back(), метод объекта History, 373, 958
background-color, свойство, 453, 458
background, CSS-свойства стиля, 459
baseURI, свойство объекта Node, 998
before(), метод объекта jQuery, 572
BeforeUnloadEvent, объект, 886
beginPath(), метод объекта CanvasRenderingContext2D, 674, 897
behavior, свойство, 641
bezierCurveTo(), метод объекта CanvasRenderingContext2D, 685, 897
bind(), метод, 211
  Function, объект, 231, 812
  и динамические события, 584
  метод Function.bind() для ECMAScript 3, 212
  регистрация обработчиков событий в библиотеке jQuery, 579
_blank, имя окна, 383
Blob, объект, 732, 886
BlobBuilder, объект, 736, 887
blur, событие, 433, 480
  всплывающая альтернатива, 485
blur(), метод
  Element, объект, 929
  Window, объект, 1026
<body>, элемент, установка обработчиков событий, 490
body, свойство объекта HTMLDocument, 396
Boolean, объект, 779
  toString(), метод, 780
  valueOf(), метод, 780
Boolean(), конструктор, 64
Boolean(), функция, преобразование типов, 69

border, свойство, 447
bottom, top, right и left, свойства стиля, 423, 456, 457
box-sizing, свойство, 457
break, инструкция, 125
break, ключевое слово, 118
browser, свойство, 610
btoa(), метод объекта Window, 1026
bubbles, свойство объекта Event, 937
buffer, свойство объекта ArrayBufferView, 885
buffered, свойство объекта MediaElement, 661, 988
bufferedAmount, свойство объекта WebSocket, 1021
<button>, элемент, 428
button, свойство объекта Event, 501, 502, 937
buttons, свойство объекта Event, 942
byteLength, свойство объекта ArrayBufferView, 885
byteOffset, свойство объекта ArrayBufferView, 885

C

<canvas>, элемент, 672
CACHE, раздел в файле объявления кэшируемого приложения, 645
call() и apply(), методы объекта Function, 193, 210
    ограничение на использование в безопасных подмножествах, 293
call(), метод объекта Function, 159, 812
callee, свойство объекта Arguments, 761, 762
callee и caller, свойства
    Arguments, объект, 196
    ограничение на использование в безопасных подмножествах, 293
caller, свойство объекта Function, 813
cancelable, свойство объекта Event, 937
cancelBubble, свойство объекта Event, 938
cancelBubble(), метод объекта Event, 584
canPlayType(), метод объекта MediaElement, 991
CanvasGradient, объект, 687, 690, 890
    addColorStop, метод, 890
    создание, 898
CanvasPattern, объект, 687, 890
    создание, 898
CanvasRenderingContext2D, объект, 673, 890
    arc(), метод, 896
    arcTo(), метод, 897
    beginPath(), метод, 897
    bezierCurveTo(), метод, 897
    clearRect(), метод, 897
    clip(), метод, 691, 693, 897
    closePath(), метод, 898
    createImageData(), метод, 898
    createLinearGradient(), метод, 690, 898
    createPattern(), метод, 689, 898
    createRadialGradient(), метод, 690, 898
    drawImage(), метод, 696, 899
    fill(), метод, 899
    fillRect(), метод, 899
    fillText(), метод, 692, 899
    getImageData(), метод, 900
    isPointInPath(), метод, 900
    lineTo(), метод, 901
    measureText(), метод, 693, 901
    moveTo(), метод, 901
    putImageData(), метод, 901
    quadraticCurveTo(), метод, 902
    rect(), метод, 902
    restore(), метод, 691, 902
    rotate(), метод, 902
    save() и restore(), методы, 894
    save(), метод, 691, 902
    scale(), метод, 902
    setTransform(), метод, 903
    stroke(), метод, 691, 903
    strokeRect(), метод, 903
    strokeText(), метод, 692, 903
    transform(), метод, 903
    графические атрибуты, 678
    графические атрибуты эффекта тени, 695
    композиция, 698
    рисование и заливка кривых, 685
    рисование прямоугольников, 687
Canvas, объект
    getContext(), метод, 889
    toDataURL(), метод, 889
    манипулирование пикселами, 894
    массив байтов в свойстве CanvasPixelArray, 730
    справка, 888
Canvas, прикладной интерфейс, 673
    Canvas, объект, 673
        getContext(), метод, 673
        toDataURL(), метод, 698
    атрибуты рисования линий, 691
    изображения, 696
    композиция, 698
    контур, 673
    отсечение, 693
    преобразование системы координат, 680
    пример рисования красного квадрата и голубого круга, 673
    прямоугольники, 687
    размеры и система координат холста, 679
    рисование и заливка кривых, 685
    рисование линий и заливка многоугольников, 675
    рисование текста, 692
    цвет, прозрачность, градиенты и шаблоны, 687
    эффект тени, 695
caption, свойство объекта Table, 1011
case, ключевое слово, 117
case, конструкции в инструкции switch
    завершение ключевым словом break, 118
catch, конструкция (инструкция try/catch/finally), 129
    множественные блоки catch, 309
cd(), метод объекта ConsoleCommandLine, 907
CDATASection, узлы, 411
ceil(), функция, объект Math, 824
cellIndex, свойство объекта TableCell, 1013
cells, свойство объекта TableRow, 1013
CERT Advisory, о межсайтовом скриптинге, 367
change(), метод объекта jQuery, 575
changedTouches, свойство события прикосновения, 489
char, свойство объекта Event, 486, 942
CharacterData, объект, 412
characterSet, свойство объекта Document, 915
charAt(), метод объекта String, 864
charCode, свойство объекта Event, 938
charCodeAt(), метод объекта String, 535, 864
charset, свойство
    Document, объект, 915
    Script, объект, 1007

checked, свойство
  Input, объект, 964
  элементов форм, 431
checkValidity(), метод объекта Form, 951
child, свойства объекта Element, 402
childElementCount, свойство объекта Element, 402, 926
childNodes, свойство объекта Node, 401, 998
children, свойство объекта Element, 402, 405, 926
children(), метод объекта jQuery, 620
Chrome, веб-броузер, 504
  textInput, событие, 486
  реализация прикладного интерфейса к файловой системе, 742
  текущая версия, 355
class, атрибут, 31, 159, 160
  HTML-элементов, 381, 397, 406
  проверка на принадлежность объекта классу Function, 214
classList, свойство, 470, 567
  Element, объект, 926
  реализация функциональности (пример), 471
className, свойство, 334, 397, 470
  Element, объект, 334, 926
clear(), метод объекта Storage, 1009
clearData(), метод объекта DataTransfer, 912
clearInterval(), метод, 370
  Window, объект, 1026
  WorkerGlobalScope, объект, 1031
clearInterval(), функция, 323
clearQueue(), метод объекта jQuery, 595
clearRect(), метод объекта CanvasRenderingContext2D, 687, 897
clearTimeout(), метод
  Window, объект, 1026
  WorkerGlobalScope, объект, 1031
clearTimeout(), функция, 323
clearWatch(), метод объекта Geolocation, 708, 955
click, событие, 500
  detail, свойство, 483
  регистрация обработчика, 31
  регистрация обработчиков событий в элементах button, 491
click(), метод
  Element, объект, 929
  jQuery, объект, 575
client, свойства элементов документа, 426
clientHeight и clientWidth, свойства объекта Element, 423, 926
clientLeft и clientTop, свойства объекта Element, 926
ClientRect, объект, 904
clientX и clientY, свойства объекта Event, 500, 501, 938
clip, свойство, 459
clip(), метод объекта CanvasRenderingContext2D, 691, 693, 897
clone(), метод объекта jQuery, 573
cloneNode(), метод объекта Node, 413, 999
close(), метод
  Document, объект, 918
  EventSource, объект, 943
  MessagePort, объект, 994
  WebSocket, объект, 756, 1021
  Window, объект, 1026
  WorkerGlobalScope, объект, 722, 1031
  генераторов, 305
  объектов Window и Document, 385
closed, свойство объекта Window, 385
CloseEvent, объект, 904
closePath(), метод объекта CanvasRenderingContext2D, 675, 898
closest(), метод объекта jQuery, 621
Closure, фреймворк, 368
code, свойство
  DOMException, объект, 923
  FileError, объект, 946
  GeolocationError, объект, 957
  MediaError, объект, 992
colorDepth, свойство объекта Screen, 1006
colSpan, свойство объекта TableCell, 1013
Comet, архитектура, 525
  на основе стандарта Server-Sent Events, 550
  транспортные механизмы, 526
compareDocumentPosition(), метод объекта Node, 1000
compareTo(), метод, реализация в классах, 246
compatMode, свойство объекта Document, 358, 915
complete, свойство объекта Image, 963
complete, свойство объекта с параметрами анимационного эффекта, 591
concat(), метод
  Array, объект, 173, 765
  String, объект, 865
configurable, атрибут, 138
  удаление свойств, 147
confirm(), метод объекта Window, 377
Console, объект, 905
  методы, 905
ConsoleCommandLine, объект, 907
console.log(), функция, 727
const, ключевое слово, 295
constructor, свойство, 158
  имя конструктора как идентификатор класса, 234
  ограничение на использование в безопасных подмножествах, 293
contains(), метод объекта DOMTokenList, 470, 924
contains(), функция, 610
content-box, модель, 457
contentDocument, свойство объекта IFrame, 961
contenteditable, свойство объекта Element, 441
contents(), метод объекта jQuery, 620
Content-Type, заголовок
  HTTP-запросы, 530
  автоматическая установка в запросе объектом XMLHttpRequest, 540
  переопределение некорректного MIME-типа в ответе, 535
contentType, параметр, 600
contentWindow, свойство объекта IFrame, 386, 961
context, свойство, 969
  jQuery, объект, 562
contextmenu, событие, 500, 501
continue, инструкция, 126
  без метки, 127
control, свойство объекта Label, 984
controls, атрибут элементов <audio> и <video>, 658
controls, свойство объекта MediaElement, 988
cookie, свойство объекта Document, 437, 915
cookies, 634, 635
  атрибуты, определяющие срок хранения и область видимости, 635

  как определить, когда поддерживаются, 635
  ограничения на размер и количество, 639
  реализация хранилища, 639
  сохранение, 637
  чтение, 638
cookiesEnabled(), метод объекта Navigator, 376
coords, свойство объекта Geoposition, 957
CORS (Cross-Origin Resource Sharing), заголовки, 545
cos(), функция, объект Math, 825
count(), метод объекта Console, 906
create(), функция, 141, 841
  добавление свойств во вновь созданный объект, 156
createCaption(), метод объекта Table, 1012
createComment(), метод объекта Document, 918
createDocument(), метод объекта DOMImplementation, 923
createDocumentFragment(), метод объекта Document, 416, 918
createDocumentType(), метод объекта DOMImplementation, 923
createElement(), метод объекта Document, 413, 918
createElementNS(), метод объекта Document, 413, 669, 918
createEvent(), метод объекта Document, 918
createHTMLDocument(), метод объекта DOMImplementation, 923
createImageData(), метод объекта CanvasRenderingContext2D, 898
createIndex(), метод хранилища объектов, 750
createLinearGradient(), метод объекта CanvasRenderingContext2D, 690, 898
createObjectStore(), метод объекта IndexedDB, 750
createObjectURL(), метод объекта URL, 1019
createObjectURL(), функция, 737
createPattern(), метод объекта CanvasRenderingContext2D, 689, 898
createProcessingInstruction(), метод объекта Document, 918
createRadialGradient(), метод объекта CanvasRenderingContext2D, 690, 898
createRange(), метод, 440
createStyleSheet(), метод объекта Document, 475
createTBody(), метод объекта Table, 1012
createTextNode(), метод объекта Document, 413, 918
createTFoot(), метод объекта Table, 1012
createTHead(), метод объекта Table, 1012
css(), метод объекта jQuery, 566
CSS2Properties, объект, 908
CSS Animations, модуль, 450
CSS (Cascading Style Sheets – каскадные таблицы стилей), 444
  блочная модель и детали позиционирования, 455
  выбор элементов документа по классу CSS, 397
  использование в JavaScript для оформления документов HTML, 31
  наиболее важные свойства стиля, 450
  обзор, 445
  определение стилей в веб-странице (пример), 448
  определение стилей для объектов Element, 334
  отображение и видимость элементов, 457
  перекрытие полупрозрачных окон (пример), 460
  позиционирование элементов, 451
  получение вычисленных стилей, 468
  селекторы, 398
  ультрасовременные свойства, 450
  управление CSS-классами, 470
  управление встроенными стилями, 463
    анимационные эффекты, 465
  управление таблицами стилей, 472
  цвет, прозрачность и полупрозрачность, 458
  частичная видимость, свойства overflow и clip, 459
  чтение и запись значений CSS-атрибутов методами объекта jQuery, 566
  чтение и запись значений CSS-классов методами объекта jQuery, 566
CSSOM-View Module, стандарт, 421
CSSRule, объект, 473, 908
cssRules, свойство объекта CSSStyleSheet, 473, 910
CSSStyleDeclaration, объект, 909
  вычисленные стили, 468
  использование для управления встроенными стилями, 463
  описание стилей, связанных с селектором, 473
  свойства, соответствующие сокращенным формам записи, 464
CSSStyleSheet, объект, 472, 910
  disabled, свойство, 473
  получение, вставка и удаление правил из таблиц стилей, 473
  создание, 475
cssText, свойство
  CSSRule, объект, 909
  CSSStyleDeclaration, объект, 465, 474, 910
CSS Transforms, модуль, 450
CSS Transitions, модуль, 467, 468
ctrlKey, свойство, 500, 519
  Event, объект, 938
  событий мыши, 483
currentSrc, свойство объекта MediaElement, 988
currentStyle, свойство (IE), 469
currentStyle, свойство объекта Element, 926
currentTarget, свойство объекта Event, 578, 938
currentTime, свойство объекта MediaElement, 660, 988
customError, свойство объекта FormValidity, 954

D

\\d (цифры, ASCII), 279
\\D (не цифры, ASCII) любой символ, не являющийся цифровым, 279
data://, URL-адрес, 949
data, свойство
  CharacterData, объект, 412
  Comment, объект, 905
  Event, объект, 515, 518, 579, 942
  ImageData, объект, 730, 963
  MessageEvent, объект, 717, 993
  ProcessingInstruction, объект, 1004
data(), метод объекта jQuery, 570, 624
dataset, свойство объекта Element, 407, 926
DataTransfer, объект, 509, 511, 911
  files, свойство, 539, 735
dataTransfer, свойство объекта Event, 509, 511, 938
DataView, объект, 913
  методы чтения/записи значений из ArrayBuffer, 731

Date(), конструктор, 781
datepicker(), метод, 626
dblclick, событие, 500, 501
dblclick(), метод объекта jQuery, 575
debug(), метод объекта Console, 906
debugger, инструкция, 132
decodeURI(), функция, 801
decodeURIComponent(), функция, 372, 637, 802
defaultCharset, свойство объекта Document, 916
defaultChecked, свойство
  Checkbox, объект, 434
  Input, объект, 964
defaultPlaybackRate, свойство объекта MediaElement, 660, 989
defaultPrevented, свойство объекта Event, 497, 938
defaultSelected, свойство объекта Option, 1002
defaultValue, свойство
  Input, объект, 964
  Output, объект, 1003
  TextArea, объект, 1016
defaultView, свойство объекта Document, 916
defer, свойство объекта Script, 1007
__defineGetter__() и __defineSetter__(), методы, 158, 405
defineProperties(), функция, 842
  определение и изменение нескольких свойств сразу, 156
defineProperty(), функция, 124, 155, 158, 842
  обеспечение доступа только для чтения к свойству length массивов, 168
  определение настраиваемых свойств, доступных только для чтения, 146
  определение неперечислимых свойств, 262
delay(), метод объекта jQuery, 593
delegate(), метод объекта jQuery, 585
delete, оператор, 85, 107
  побочные эффекты, 87, 110
  удаление атрибутов в расширении E4X, 313
  удаление свойств, 147
  удаление свойств глобального объекта, 148
  удаление элементов массивов, 169
deleteCaption(), метод объекта Table, 1012
deleteCell(), метод объекта TableRow, 1013
deleteData(), метод
  Comment, узел, 905
  Text, объект, 1015
deleteRow(), метод
  TableSection, объект, 1014
  Table, объект, 1012
deleteRule(), метод объекта CSSStyleSheet, 474, 911
deleteTFoot(), метод объекта Table, 1012
deleteTHead(), метод объекта Table, 1012
deltaMode, свойство объекта Event, 942
deltaX, deltaY и deltaZ, свойства объекта Event, 505, 506, 942
dequeue(), метод объекта jQuery, 594
designMode, свойство объекта Document, 441, 916
detach(), метод объекта jQuery, 574
detachEvent(), метод, 492, 945
detail, свойство объекта Event, 505, 506, 938
DHTML (Dynamic HTML – динамический HTML), 336
dialogArguments, свойство объекта Window, 378, 1023
die(), метод объекта jQuery, 586
dir, свойство объекта Document, 916
dir(), метод
  ConsoleCommandLine, объект, 907
  Console, объект, 906
DirectoryEntry, объект, 743
DirectoryReader, объект, 743
dirxml(), метод
  ConsoleCommandLine, объект, 907
  Console, объект, 906
disabled, свойство
  CSSStyleSheet, объект, 910
  FieldSet, объект, 945
  FormControl, объект, 952
  Link, объект, 985
  Option, объект, 1002
  Style, объект, 1011
dispatchEvent(), метод объекта EventTarget, 944
dispatchFormChange(), метод объекта Form, 951
dispatchFormInput(), метод объекта Form, 951
display, свойство, 457
DOCTYPE, объявление
  режим совместимости и стандартный режим, 358
doctype, свойство объекта Document, 916
Document, объект, 334, 390, 914
  addEventListener(), метод, 492
  close(), метод, 385
  compatMode, свойство, 358
  cookie, свойство, 634
    анализ содержимого, 638
  createDocumentFragment(), метод, 416
  createElementNS(), метод, 413
  createElement(), метод, 413
  createStyleSheet(), метод, 475
  createTextNode(), метод, 413
  designMode, свойство, 441
  domain, свойство, 364
  elementFromPoint(), метод, 424
  execCommand(), метод, 442
  forms, свойство, 430
  getElementById(), метод, 381, 393, 430
  getElementByTagName(), метод, 395
  getElementsByClassName(), метод, 398
  getElementsByTagName(), метод, 430
  importNode(), метод, 413
  location, свойство, 371
  open(), метод, 374
  queryCommandEnabled(), метод, 442
  queryCommandIndeterm(), метод, 443
  queryCommandState(), метод, 442
  queryCommandValue(), метод, 442
  querySelector(), метод, 399
  querySelectorAll(), метод, 399, 473
  readyState, свойство, 351, 498
  removeEventListener(), метод, 492
  styleSheets, свойство, 473
  URL, свойство, 371
  write(), метод, 345, 351, 374, 438
  writeln(), метод, 439
  как HTTP-ответ, 534
  методы, 917
  свойства, 437, 915
  события, 921
document, свойство объекта Window, 334, 1023
document.all[], коллекция, 400
documentElement, свойство объекта Document, 396, 422, 916

DocumentFragment, объект, 393, 416, 921
  querySelector(), метод, 399
  querySelectorAll(), метод, 399
  реализация метода insertAdjacentHTML() с использованием свойства innerHTML, 417
  реализация перестановки дочерних узлов в обратном порядке, 417
  создание, 918
DocumentType, объект, 922
Dojo, фреймворк, 368
DOM (Document Object Model – объектная модель документа)
  DOM Level 3 Events, спецификация, 485
    предлагаемые к реализации методы, 942
    предлагаемые к реализации свойства, 941
  textinput, событие, 516
  обзор, 390
  реализация типов как классов, 404
domain, свойство объекта Document, 364, 437, 916
DOMContentLoaded, событие, 351, 498, 499
DOMException, объект, 922
DOMImplementation, объект, 923
DOMMouseScroll, событие, 504
DOMSettableTokenList, объект, 923
DOMTokenList, объект, 470, 924
  add(), метод, 470
  contains(), метод, 470
  remove(), метод, 470
  toggle(), метод, 470
do/while, циклы, 120
  continue, инструкция, 127
draggable, атрибут, 509
drawImage(), метод объекта CanvasRenderingContext2D, 696, 899
dropEffect, свойство объекта DataTransfer, 509, 912
dropzone, атрибут, 511
duration, свойство объекта MediaElement, 660, 989

E

E, константа (объект Math), 825
E4X (ECMAScript for XML), расширение
  введение, 310
each(), метод объекта jQuery, 563
each(), функция, 610
ECMAScript, стандарт расширения JavaScript, 290
ECMAScript 5, стандарт
  зарезервированные слова, 44
  классы
    дескрипторы свойств, 268
    определение неизменяемых классов, 263
    определение неперечислимых свойств, 262
    подклассы, 267
    предотвращение расширения классов, 266
    сокрытие данных объекта, 265
  методы массивов, 176, 563
effectAllowed, свойство объекта DataTransfer, 509, 511, 912
Element, класс, 393
Element, объект, 334, 925
  attributes, свойство, 408
  contenteditable, свойство, 441
  dataset, свойство, 407
  getAttribute(), метод, 406
  getAttribute() и setAttribute(), методы, 465
  getBoundingClientRect(), метод, 426
  getClientRects(), метод, 424
  hasAttribute(), метод, 407
  innerHTML, свойство, 409
  insertAdjacentHTML(), метод, 410
  Node, объекты, 401
  offsetLeft и offsetTop, 426
  offsetParent, свойство, 426
  offsetWidth и offsetHeight, 426
  outerHTML, свойство, 409
  removeAttribute(), метод, 407
  setAttribute(), метод, 406
  методы, 929
  обработчики событий, 931
  определение собственных методов, 404
  свойства, 925
elementFromPoint(), метод объекта Document, 424, 918
elements, свойство
  FieldSet, объект, 945
  Form, объект, 950
  элементов форм, 430
else, конструкция во вложенных инструкциях if, 115
else if, инструкция, 116
<embed>, элемент, 571
embeds, свойство
  Document, объект, 916
  HTMLDocument, объект, 396
empty(), метод объекта jQuery, 574
enableHighAccuracy, параметр объекта Geolocation, 956
encodeURI(), функция, 802
encodeURIComponent(), функция, 637, 803
encoding, свойство объекта Form, 431
enctype, свойство объекта Form, 950
end(), метод
  jQuery, объект, 621
  TimeRanges, объект, 1017
ended, свойство объекта MediaElement, 989
enumerable, атрибут (свойств), 154, 155
  propertyIsEnumerable(), функция проверки, 148
eq(), метод объекта jQuery, 618
equals(), метод, реализация в классах, 244
error, свойство
  FileReader, объект, 947
  MediaElement, объект, 662, 663, 989
error, событие
  FileReader, объект, 740
Error(), конструктор, 804
error(), метод
  Console, объект, 906
  jQuery, объект, 575
ErrorEvent, объект, 935
escape(), функция, 806
eval(), функция, 102, 807
  отсутствует в безопасных подмножествах, 292
EvalError, объект, 808
Event, объект, 577, 936
  defaultPrevented, свойство, 497
  preventDefault(), метод, 497
  returnValue, свойство, 497
  stopImmediatePropagation(), метод, 498
  stopPropagation(), метод, 497
  в библиотеке jQuery, 577

- константы, определяющие значения свойства eventPhase, 937
- методы, 941
- предлагаемые к реализации методы, 942
- предлагаемые к реализации свойства, 941
- свойства, 937

event, свойство объекта Window, 493, 1023
eventPhase, свойство объекта Event, 939
- константы, определяющие возможные значения, 937

EventSource, объект, 550, 943
- имитация с помощью объекта XMLHttpRequest, 552
- использование в простом клиенте чата, 551
- константы, определяющие допустимые значения свойства readyState, 943

EventTarget, объект, 944
every(), метод объекта Array, 178, 765
excanvas.js, библиотека, 356
exec(), метод объекта RegExp, 288, 858
execCommand(), метод объекта Document, 442, 918
exp(), функция, объект Math, 825
expires, свойство, сохранение данных с помощью механизма userData, 641
ExplorerCanvas, проект, 672
extend(), функция, 150, 611
extensible, атрибут, 160, 161

F

fadeTo(), методы объекта jQuery, 588
FALLBACK, раздел в файле объявления кэшируемого приложения, 645
false и true, значения, 67
FieldSet, объект, 945
File, объект, 734, 945
File API, спецификация, 487
FileEntry, объект, 743
FileError, объект, 946
FileList, объект, 735
filename, свойство объекта ErrorEvent, 936
FileReader, объект, 734, 946
- возбуждение событий, 487
- константы, определяющие возможные значения свойства readyState, 947
- методы, 947
- обработчики событий, 948
- свойства, 947
- слежение за ходом выполнения асинхронных операций ввода/вывода, 487
- чтение двоичных объектов, 739

FileReaderSync, объект, 741, 949
files, свойство
- DataTransfer, объект, 512, 735, 912
- Input, объект, 964

FileUpload объект, ограничение возможностей из соображений безопасности, 362
FileWriter, объект, 734, 743
fill(), метод объекта CanvasRenderingContext2D, 674, 899
fillRect(), метод объекта CanvasRenderingContext2D, 687, 899
fillStyle, свойство объекта CanvasRenderingContext2D, 678, 894
fillText(), метод объекта CanvasRenderingContext2D, 692, 899
filter, свойство (IE), 459
filter(), метод
- Array, объект, 177, 766
- jQuery, объект, 618

finally, конструкция (инструкция try/catch/finally), 129
find(), метод объекта jQuery, 620
Firefox, веб-броузер, 375
- charCode, свойство, 515
- DOMMouseScroll, событие, 504
- версии и расширения JavaScript, 290, 297
- глобальный подход к композиции, 700
- изменения в прикладном интерфейсе объекта History в Firefox 4, 713
- текущая версия, 355

:first-line и :first-letter, псевдоэлементы (CSS), 400
first(), метод объекта jQuery, 618
firstChild, свойство объекта Node, 401, 998
firstElementChild, свойство объекта Element, 402, 927
Float32Array, класс, 728, 1017
Float64Array, класс, 728, 1017
floor(), функция, объект Math, 826
fn, объект-прототип (jQuery), 622
focus, событие, 433, 480
- focusin и focusout, всплывающие версии, 485
- фокус ввода в элементах документа, 485

focus(), метод
- Element, объект, 929
- Window, объект, 1026

font, свойство, 447, 892, 894
- текста в холсте, 692

fontFamily, свойство, 469
forEach(), метод объекта Array, 171, 176, 767
for/each, циклы, 300
- в генераторах массивов, 307
- в расширении E4X, 312

for/in, циклы, 121, 122
- continue, инструкция, 127
- в генераторах массивов, 307
- вызов метода jQuery.each(), 563
- использование ключевого слова let для инициализации переменной цикла, 295
- использование с ассоциативными массивами, 144
- итерации по полям, методам и свойствам Java-классов и объектов, 317
- обход элементов массива, 170, 171
- объявление переменных с помощью ключевого слова let, 295
- перечисление свойств, 149
- порядок перечисления свойств, 123
- расширение для работы с итерируемыми объектами, 302

Form, объект, 949
- checked, свойство, 431
- elements, свойство, 430
- submit() и reset(), методы, 432
- value, свойство, 431
- методы, 951
  - обработчики событий, 951
- свойства, 950

form, свойство, 432, 433
- FormControl, объект, 952
- Label, объект, 984
- Meter, объект, 995
- Option, объект, 1002
- Progress, объект, 1005

formAction, свойство
  Button, объект, 888
  Input, объект, 965
FormControl, объект, 951
  обработчики событий, 953
  свойства, 952
FormData, объект, 540, 953
formEnctype, свойство
  Button, объект, 888
  Input, объект, 965
formMethod, свойство
  Button, объект, 888
  Input, объект, 965
formNoValidate, свойство
  Button, объект, 888
  Input, объект, 965
forms, свойство объекта Document, 430, 916
forms, свойство объекта HTMLDocument, 396
formTarget, свойство
  Button, объект, 888
  Input, объект, 965
FormValidity, объект, 954
forward(), метод объекта History, 373, 958
frameElement, свойство объекта Window, 386, 1023
frames, свойство объекта Window, 386, 1023
<frameset> и <frame>, элементы (устаревшие), 382
freeze(), функция, 161, 843
fromCharCode(), метод объекта String, 865
fromElement, свойство объекта Event, 939
fs, модуль для работы с файлами и файловой системой, Node, 325
function, инструкция, 113
Function, класс, 50
  toString(), метод, 71
function, ключевое слово, 82, 185
  создание переменной, 387
Function, объект, 809
  apply(), метод, 811
  arguments, свойство, 811
  bind(), метод, 211, 231, 812
  caller, свойство, 813
  call(), метод, 159, 812
  length, свойство, 814
  prototype, свойство, 814
  toString(), метод, 213, 814
  методы, 810
  определение собственных свойств, 200
  свойства, 810
Function(), конструктор, 213
  отсутствует в безопасных подмножествах, 292

G

g, флаг (регулярные выражения), 285, 286, 289
Geocoordinates, объект, 955
Geolocation, объект, 707, 955
  демонстрация всех возможностей, 710
  использование для отображения карты, 708
geolocation, свойство объекта Navigator, 376, 708, 996
GeolocationError, объект, 956
Geoposition, объект, 957
GET, метод, 529
  выполнение HTTP-запросов с данными в формате HTML-форм, 537
  запрос без тела, 530
get(), функция, 600
getAllResponseHeaders(), метод объекта XMLHttpRequest, 1036
getAttribute(), метод объекта Element, 406, 929
getAttributeNS(), метод объекта Element, 929
getBoundingClientRect(), метод объекта Element, 423, 426, 929
getClientRects(), метод объекта Element, 424, 929
getComputedStyle(), метод объекта Window, 468, 1026
getContext(), метод объекта Canvas, 673, 889
getCurrentPosition(), метод объекта Geolocation, 708, 955
getData(), метод объекта DataTransfer, 511, 912
getDate(), метод объекта Date, 785
getDay(), метод объекта Date, 785
getElementById(), метод объекта Document, 381, 393, 430, 919
getElementByName(), метод объекта HTMLDocument, 394
getElementByTagName(), метод объекта Document, 395
getElementsByClassName(), метод
  Document, объект, 398, 919
  Element, объект, 929
getElementsByName(), метод объекта Document, 919
getElementsByTagName(), метод
  Document, объект, 430, 919
  Element, объект, 929
getElementsByTagNameNS(), метод
  Document, объект, 919
  Element, объект, 930
  выбор форм и элементов форм, 430
getFloat32(), метод объекта DataView, 913
getFloat64(), метод объекта DataView, 913
getFullYear(), метод объекта Date, 785
getHours(), метод объекта Date, 786
getImageData(), метод объекта CanvasRenderingContext2D, 900
getInt8(), метод объекта DataView, 914
getInt16(), метод объекта DataView, 913
getInt32(), метод объекта DataView, 914
getItem(), метод объекта Storage, 1009
getJSON(), функция, 598
getMilliseconds(), метод объекта Date, 786
getMinutes(), метод объекта Date, 786
getModifierState(), метод объекта Event, 943
getMonth(), метод объекта Date, 786
getOwnPropertyDescriptor(), функция, 158, 844
getOwnPropertyNames(), функция, 151, 845
getPrototypeOf(), функция, 158, 845
getResponseHeader(), метод объекта XMLHttpRequest, 1037
getScript(), функция, 598
getSeconds(), метод объекта Date, 786
getSelection(), метод объекта Window, 440
getTime(), метод объекта Date, 787
getTimezoneOffset(), метод объекта Date, 787
getUint8(), метод объекта DataView, 914
getUint16(), метод объекта DataView, 914
getUint32(), метод объекта DataView, 914
getUTCDate(), метод объекта Date, 787
getUTCDay(), метод объекта Date, 788
getUTCFullYear(), метод объекта Date, 788
getUTCHours(), метод объекта Date, 788

getUTCMilliseconds(), метод объекта Date, 788
getUTCMinutes(), метод объекта Date, 788
getUTCMonth(), метод объекта Date, 789
getUTCSeconds(), метод объекта Date, 789
getYear(), метод объекта Date, 789
global, свойство объекта RegExp, 287, 858, 859
globalAlpha, свойство, 688, 894
globalCompositeOperation, свойство, 698, 893, 895
globalEval(), функция, 611
GMT (Greenwich Mean Time – среднее время по Гринвичу), 781
go(), метод объекта History, 373, 958
Google, компания, предоставляет возможность загрузки сценариев веб-приложениями с других сайтов, 340
grep(), метод объекта jQuery, 619
grep(), функция, 611
group(), метод объекта Console, 906
groupCollapsed(), метод объекта Console, 906
groupEnd(), метод объекта Console, 906
GWT, фреймворк, 368

H

handler, свойство объекта Event, 579
has(), метод объекта jQuery, 619
hasAttribute(), метод объекта Element, 407, 930
hasAttributeNS(), метод объекта Element, 930
hasClass(), метод объекта jQuery, 567
hasFocus(), метод объекта Document, 919
hash, свойство
  Link, объект, 985
  Location, объект, 371, 711, 986
  WorkerLocation, объект, 1033
hashchange, событие, 711
HashChangeEvent, объект, 957
hasOwnProperty(), метод класса Object, 148, 846
HAVE, константы объекта MediaElement, 988
heading, глобальная переменная, 381
heading, свойство объекта Geocoordinates, 955
HEAD, метод, получение информации о ссылках при поддержке заголовка CORS, 546
head, свойство объекта Document, 396, 916
height, свойство
  ClientRect, объект, 904
  IFrame, объект, 961
  ImageData, объект, 964
  Image, объект, 963
  Screen, объект, 377
height и width, свойства стиля, 456, 457
height(), метод объекта jQuery, 569
hide(), метод объекта jQuery, 588, 591
high, свойство объекта Meter, 995
History, объект, 373, 958
  back(), forward() и go(), методы, 373
  pushState(), метод, 712
    управление историей посещений (пример), 713
  replaceState(), метод, 713
history, свойство объекта Window, 373, 1023
host, свойство
  Link, объект, 985
  Location, объект, 371, 986
  WorkerLocation, объект, 1033
hostname, свойство
  Link, объект, 985
  Location, объект, 371, 986
  WorkerLocation, объект, 1033
:hover, псевдокласс, 656
hover(), метод объекта jQuery, 576, 580
href, атрибут элемента <a>, 342, 396
href, свойство
  CSSStyleSheet, объект, 910
  Link, объект, 985
  Location, объект, 371, 987
  WorkerLocation, объект, 1033
HSL (hue-saturation-value – тон-насыщенность-яркость), формат определения цвета, 458
HSLA (hue-saturation-value-alpha – тон-насыщенность-яркость-альфа), формат определения цвета, 458
<html>, элемент
  manifest, атрибут, 643
html(), метод объекта jQuery, 568
HTML, язык разметки
  FileUpload, объект, ограничение возможностей из соображений безопасности, 362
  <script>, элемент, 335
  Select и Option, элементы, 435
  атрибуты элементов, 405
  включение таблиц стилей в HTML-страницы, 446
  встраивание JavaScript, 337
    обработчики событий, 341
    сценарии во внешних файлах, 339
    элемент <script>, 338
  имена тегов нечувствительны к регистру символов, 395
  кнопки, 433
  непосредственное использование разметки SVG в HTML-файлах, 667
  нечувствительность к регистру, 41
  обработчики событий, 341
  переключатели и флажки, 434
  содержимое элементов в виде разметки HTML, 409
  текстовые поля ввода, 434
  управление содержимым с помощью JavaScript, 32
  формы, 428
  чтение и запись значений HTML-атрибутов методами объекта jQuery, 565
HTML5, стандарт, 706
  API механизма буксировки (drag-and-drop), 508
  classList, свойство, 470, 567
  DOMContentLoaded, событие, 498
  getElementsByClassName(), метод, 398
  innerHTML и outerHTML, свойства, 409
  insertAdjacentHTML(), метод, 410
  placeholder, атрибут текстовых полей ввода, 435
  WebSocket API, спецификация, 755
  Web Workers, спецификация, 720
  WindowProxy, объект, 389
  атрибуты с данными и свойство dataset, 407
  базы данных на стороне клиента, 747
  взаимодействие документов с разным происхождением, 716
  геопозиционирование, 707
  двоичные объекты, 732
  команды редактирования текста, 443
  кэш приложений, 643
  прикладные интерфейсы для веб-приложений, 337

  события, 486
  типизированные массивы и буферы, 728
  управление историей посещений, 711
- HTMLCollection, объект, 396, 959
  обзор, 396
  элементы форм, 430
- HTMLDocument, объект, 392, 959
  getElementByName(), метод, 394
  images, forms и links, свойства, 396
- HTMLElement, объект, 392, 959
  text, свойство, 341
  представление HTML-элементов в документе, 380
  свойства, отражающие HTML-атрибуты, 405
- htmlFor, свойство
  Label, объект, 984
  Output, объект, 1003
- HTMLFormControlsCollection, объект, 959
- HTMLOptionsCollection, объект, 960
- HTML-атрибуты, отображаемые в обработчики событий, 341
- HTML-атрибуты управления воспроизведением, 660
- HTML-формы
  FormControl, объект, 951
  FormData, объект, 953
  FormValidity, объект, 954
  Form, объект, 949
  чтение и запись значений элементов форм, 567
- HTML-элементы, 932
- HTTP, протокол, 755
  HTTP-сервер в Node (пример), 327
  взаимодействие с объектом XMLHttpRequest, 363
  модуль утилит клиента HTTP в Node, 329
  работа с помощью объекта XMLHttpRequest, 527
  работа с протоколом, 524
    использование архитектуры Comet на основе стандарта Server-Sent Events, 550
    использование элементов <script>, 548
- HTTP-методы, 529

I

- i, флаг (регулярные выражения), 285, 288
- id, атрибут HTML-элементов, 380
- id, свойство объекта Element, 927
- IDBRange, объект, 749
- if, инструкция, 114
  в генераторах массивов, 308
  вложенная, с конструкцией else, 115
- <iframe>, элемент
  и история посещений, 373
  как транспорт в архитектуре Ajax, 525
  свойства в документе для ссылки на объект Window, 395
- IFrame, объект, 961
- xignoreCase, свойство объекта RegExp, 288, 858, 860
- Image, объект, 962
- ImageData, объект, 963
  data, свойство, массив байтов, 730
  копирование в элемент <canvas>, 901
  передача фоновому потоку с помощью метода postMessage(), 725
  создание, 898
- images, свойство объекта Document, 396, 916
- <img>, элемент, 656
  как транспорт в архитектуре Ajax, 525
  отображение изображений в формате SVG, 666
- implementation, свойство объекта Document, 917
- importNode(), метод объекта Document, 413, 919
- importScripts(), метод объекта WorkerGlobalScope, 722, 1031
- in, оператор, 85, 97
  проверка существования унаследованных и неунаследованных свойств, 148
- inArray(), функция, 611
- indeterminate, свойство объекта Input, 965
- index, свойство
  Array объект, 287
  Option, объект, 1002
- index(), метод объекта jQuery, 564
- IndexedDB, прикладной интерфейс, 748
  база данных с почтовыми индексами США (пример), 750
- indexOf(), метод
  Array, объект, 180, 768
  String, объект, 866
- -Infinity (отрицательная бесконечность), 53
- Infinity (положительная бесконечность), 53
- Infinity, свойство, 817
- info(), метод объекта Console, 906
- initEvent(), метод объекта Event, 941
- initialTime, свойство объекта MediaElement, 660, 989
- innerHeight и innerWidth, свойства объекта Window, 1023
- innerHeight() и innerWidth(), методы объекта jQuery, 569
- innerHTML, свойство объекта Element, 409, 927
  использование в объектах jQuery для получения содержимого элемента, 568
- innerText, свойство объекта Element, 410
- <input>, элемент как кнопка, 433
  выгрузка файлов, 539
- input, глобальная переменная, 381
- Input, объект, 964
  методы, 966
  свойства, 964
- input, свойство, 287
- input, событие, возбуждаемое после вставки текста в элемент, 518
- inputMethod, свойство объекта Event, 515, 518, 942
- insertAdjacentHTML(), метод объекта Element, 410, 930
- insertAfter(), метод объекта jQuery, 572
- insertBefore(), метод объекта jQuery, 572
- insertBefore(), метод объекта Node, 413, 1000
- insertCell(), метод объекта TableRow, 1013
- insertData(), метод
  Comment, узел, 905
  Text, объект, 1015
- insertRow(), метод
  Table, объект, 1012
  TableSection, объект, 1014
- insertRule(), метод объекта CSSStyleSheet, 474, 911
- inspect(), метод объекта ConsoleCommandLine, 908
- instanceof, оператор, 85, 97
  и метод isPrototypeOf(), 159
  использование с конструкторами для проверки принадлежности к классу, 225

  - невозможность отличить массивы от объектов, 181
  - определение классов объектов, 232
  - работа с объектами и классами Java в Rhino, 316
- Int8Array, класс, 728, 1017
- Int16Array, класс, 728, 1017
- Int32Array, класс, 728, 1017
- Internet Explorer (IE), веб-броузер
  - API механизма буксировки (drag-and-drop), 508
  - <canvas>, элемент, 672
  - clientInformation, свойство, 374
  - currentStyle, свойство, 469
  - filter, свойство, 459
  - propertychange, событие, применение для определения факта ввода текста, 518
  - XMLHttpRequest в IE6, 528
  - блочная модель CSS, 457
  - вычисленные свойства, 469
  - и перехват событий, 496
  - методы таймера, 371
  - механизм сохранения userData, 641
  - модель событий
    - attachEvent(), метод, 349, 492, 945
    - detachEvent(), метод, 492, 945
    - события форм, 482
  - невсплывающие версии событий мыши, 484
  - не поддерживает
    - метод getElementsByClassName(), 398
  - получение выделенного текста, 440
  - свойство innerText вместо textContent, 410
  - текущая версия, 355
  - условные комментарии, 359
- iPhone и iPad, устройства компании Apple, события gesture и touch, 488
- is(), метод объекта jQuery, 564
- isArray(), метод объекта Array, 181
- isArray(), функция, 611
- isContentEditable, свойство объекта Element, 927
- isDefaultNamespace(), метод объекта Node, 1000
- isEmptyObject(), функция, 611
- isEqualNode(), метод объекта Node, 1000
- isExtensible(), функция, 160, 847
- isFinite(), функция, 55, 817
- isFrozen(), функция, 161, 847
- isFunction(), функция, 214, 611
- isNaN(), функция, 55, 817
- isPlainObject(), функция, 612
- isPointInPath(), метод объекта CanvasRenderingContext2D, 900
- isPrototypeOf(), метод, 159, 848
- isSameNode(), метод объекта Node, 1000
- isSealed(), функция, 161, 848
- isTrusted, свойство объекта Event, 939
- item(), метод
  - DOMTokenList, объект, 925
  - HTMLCollection, объект, 397, 959
  - HTMLOptionsCollection, объект, 960
  - NodeList, объект, 397, 1002
  - Select, элемент, 1009
- items, свойство объекта DataTransfer, 512
- __iterator__(), метод, 302
- Iterator(), функция, 303
  - присваивание с разложением, 303

J

- Java, язык программирования
  - изменение значений статических полей классов в Rhino, 317
  - создание экземпляров классов языка Java, 316
  - управление с помощью Rhino, 315
    - создание графического интерфейса (пример), 319
- javaEnabled(), метод объекта Navigator, 376
- javaException, свойство объекта Error, 318
- JavaScript, язык программирования
  - в веб-документах и веб-приложениях, 336
  - версии, 290, 297
  - поддержка событий колесика мыши, 504
  - справочник, 881
- javascript:, URL-адреса, 342
- join(), метод объекта Array, 172, 769
- jQuery, библиотека, 32, 622, 626, 967
  - Ajax в версии jQuery 1.5, 603
  - jQuery(), функция ($()), 558
  - анимационные эффекты, 586
  - базовые методы и свойства, 969
  - вспомогательные функции, 610
  - грамматика селекторов, 968
  - запросы и результаты запросов, 562
  - запросы на основе селекторов CSS, 400
  - имена функций и методов в официальной документации, 562
  - методы воспроизведения анимационных эффектов, 978
  - методы вставки и удаления, 974
  - методы выбора, 618
    - использование результатов выбора в качестве контекста, 619
  - методы работы с событиями, 976
  - методы работы с элементами, 972
  - обработка событий, 575
  - основы, 557
  - получение, 559
  - расширение с помощью модулей расширений, 622
  - реализация Ajax, 595
    - ajax(), функция, 601
    - get() и post(), функции, 600
    - getJSON(), функция, 598
    - getScript(), функция, 598
    - load(), метод, 595
    - вспомогательные функции, 597
    - коды состояния, 596
    - события Ajax, 608
    - типы данных, 601
  - селекторы, 613
    - группы, 617
    - комбинированные, 617
    - простые, 614
  - терминология, 561
  - фабричная функция, 967
  - функции поддержки архитектуры Ajax, 980
- jQuery, объект
  - изменение структуры документа, 571
  - методы чтения и записи, 565
- jquery, свойство, 563
- jQuery.browser, свойство, 610
- jQuery.easing, объект, 592
- jQuery.fx.speeds, объект, 586
- jQuery UI, библиотека, 625

jqXHR, объект, 603
JSON, формат представления данных, 161, 818
  jQuery.getJSON(), функция, 598
  toJSON(), метод, 162, 163
  выполнение POST-запросов с данными в формате JSON, 538
  методы, реализация в классах, 243
JSON.parse(), функция, 161, 162, 819
JSON.stringify(), функция, 161, 712, 820
JSONP, формат данных, 548
  выполнение запросов с помощью элемента <script>, 549

K
KeyboardEvent, объект, 485
key, свойство, 519
  Event, объект, 486, 942
  StorageEvent, объект, 1010
key(), метод объекта Storage, 1010
keyCode, свойство, 515, 519
  Event, объект, 939
KeyEvent, объект, 984
keyIdentifier, свойство, 519
Keymap, класс поддержки обработки комбинаций клавиш (пример), 520
keys(), метод объекта ConsoleCommandLine, 908
keys(), функция, 849
  перечисление имен свойств, 151

L
Label, объект, 984
label, свойство объекта Option, 1002
labels, свойство
  FormControl, объект, 952
  Meter, объект, 995
  Progress, объект, 1005
lang, свойство объекта Element, 927
last(), метод объекта jQuery, 618
lastChild, свойство объекта Node, 401, 998
lastElementChild, свойство объекта Element, 402, 927
lastEventId, свойство объекта MessageEvent, 993
lastIndex, свойство
  RegExp, объект, 288, 858, 860
  методы объекта String, 288
lastIndexOf(), метод
  Array, объект, 180, 769
  String, объект, 866
lastModified, свойство объекта Document, 437, 917
lastModifiedDate, свойство объекта File, 735, 946
latitude, свойство объекта Geocoordinates, 955
left, right, top и bottom, свойства объекта ClientRect, 904
left и top, свойства стиля, 452, 456
length, свойство
  Arguments, объект, 194, 209, 761, 762
  Array, объект, 764, 770
  Comment, объект, 905
  CSSStyleDeclaration, объект, 910
  DOMTokenList, объект, 924
  Form, объект, 950
  Function, объект, 814
  History, объект, 958
  HTMLCollection, объект, 959
  HTMLOptionsCollection, объект, 960
  jQuery, объект, 562
  NodeList, объект, 1002
  Select, элемент, 1008
  Storage, объект, 1009
  String, объект, 867
  Text, объект, 1014
  TimeRange, объект, 1017
  Window, объект, 387, 1023
  массивов, 166
    манипулирование, 168
    разреженных, 167
    типизированных, 1018
  функций, 209
lengthComputable, свойство объекта ProgressEvent, 1006
let, ключевое слово, 295
  использование для инициализации переменной цикла, 295
  как замена инструкции var, 295
lineCap, свойство объекта CanvasRenderingContext2D, 691, 892, 895
lineJoin, свойство объекта CanvasRenderingContext2D, 692, 892, 895
lineno, свойство объекта ErrorEvent, 936
lineTo(), метод объекта CanvasRenderingContext2D, 675, 684, 691, 901
lineWidth, свойство объекта CanvasRenderingContext2D, 678, 691, 892, 895
Link, объект, 984
links, свойство объекта Document, 396, 917
list, свойство объекта Input, 965
live(), метод объекта jQuery, 585
LN2, константа (объект Math), 826
LN10, константа (объект Math), 826
load, событие, 345, 482
  FileReader, объект, 740
  onload, обработчик события объекта Window, 335
  документ, распространение событий, 496
  поддержка в веб-броузерах, 351
  регистрация обработчика, 31
load(), метод
  jQuery, объект, 575
  MediaElement, объект, 992
  мультимедийных элементов, 660
  утилита Ajax в библиотеке jQuery, 595
load(), функция (Rhino), 315
loaded, свойство объекта ProgressEvent, 542, 1006
locale, свойство объекта Event, 942
localeCompare(), метод объекта String, 867
localName, свойство объекта Attr, 885
localName, свойство объекта Element, 927
localStorage, свойство объекта Window, 630, 1023
  автономные веб-приложения, 650
  в сравнении с cookies, 636
  прикладной программный интерфейс объекта Storage, 632
  события, 633
  срок хранения и область видимости, 630
Location, объект, 986
  assign(), метод, 372
  hash, свойство, 371, 711
  hostname, свойство, 371
  host, свойство, 371
  href, свойство, 371
  pathname, свойство, 371
  port, свойство, 371
  protocol, свойство, 371

- reload(), метод, 373
- replace(), метод, 372
- search, свойство, 371
- toString(), метод, 371

- location, свойство
  - Document, объект, 371, 437, 917
  - Event, объект, 942
  - Window, объект, 334, 371, 1024
  - WorkerGlobalScope, объект, 724, 1031
- LOG2E, константа (объект Math), 827
- LOG10E, константа (объект Math), 827
- log(), метод объекта Console, 727, 906
- log(), функция, объект Math, 827
- longitude, свойство объекта Geocoordinates, 955
- __lookupGetter__() и __lookupSetter__(), методы, 158
- lookupNamespaceURI(), метод объекта Node, 1001
- lookupPrefix(), метод объекта Node, 1001
- loop, свойство объекта MediaElement, 661, 989
- low, свойство объекта Meter, 995

M

- m, флаг (регулярные выражения), 285, 288
- makeArray(), функция, 612
- map(), метод
  - Array, объект, 177, 215, 770
  - объектов Array и jQuery, 564
- map(), функция, 612
- margin, свойство, 464, 465
- match(), метод объекта String, 286, 288, 868
- Math, объект, 822
  - abs(), функция, 823
  - acos(), функция, 823
  - asin(), функция, 823
  - atan2(), функция, 824
  - atan(), функция, 824
  - ceil(), функция, 824
  - cos(), функция, 825
  - exp(), функция, 825
  - floor(), функция, 826
  - log(), функция, 827
  - max(), функция, 210, 827
  - min(), функция, 828
  - pow(), функция, 828
  - random(), функция, 829
  - round(), функция, 829
  - sin(), функция, 829
  - sqrt(), функция, 829
  - tan(), функция, 830
  - константы, 822
  - статические функции, 822
  - функции и константы как свойства, 53
- max, свойство
  - Input, объект, 965
  - Meter, объект, 995
  - Progress, объект, 1005
- max(), функция объекта Math, 210, 827
- max-age, атрибут cookies, 636
- maximumAge, параметр объекта Geolocation, 956
- maxLength, свойство объекта Input, 965
- MAX_VALUE, константа (объект Number), 833
- measureText(), метод объекта CanvasRenderingContext2D, 693, 901
- media, свойство
  - CSSStyleSheet, объект, 910
  - Style, объект, 1011
- MediaElement, объект, 987
  - константы, определяющие возможные значения свойства readyState, 987
  - методы, 991
  - обработчики событий, 990
  - свойства, 988
- MediaError, объект, 992
- merge(), функция, 612
- message, свойство
  - Error, объект, 804, 805
  - ErrorEvent, объект, 936
  - EvalError, объект, 809
  - GeolocationError, объект, 957
  - ReferenceError, объект, 857
  - URIError, объект, 880
- message, событие, 717
- MessageChannel, объект, 724, 992
- MessageEvent, объект, 993
- MessagePort, объект, 724, 994
- metaKey, свойство, 500, 519
  - Event, объект, 939
    - в библиотеке jQuery, 578
  - событий мыши, 483
- Meter, объект, 995
- method, свойство объекта Form, 431, 950
- MIME-тип
  - данных в формате HTML-форм, 536
  - объявления кэшируемого приложения, 644
  - определение в HTTP-заголовках Content-Type, 530
  - переопределение некорректного типа в HTTP-ответе, 535
- MIN_VALUE, константа (объект Number), 833
- min, свойство
  - Input, объект, 965
  - Meter, объект, 995
- min(), функция, объект Math, 828
- miterLimit, свойство, 895
  - CanvasRenderingContext2D, объект, 692
- monitorEvents(), метод объекта ConsoleCommandLine, 908
- mousedown, событие, 500
- mouseenter, событие, 500
- MouseEvent, объект, 996
- mouseleave, событие, 500
- mousemove, событие, 500
- mouseout, событие, 500
- mouseover, событие, 500
- mouseup, событие, 500
- mousewheel, событие, 484, 485
- moveTo(), метод объекта CanvasRenderingContext2D, 675, 901
- Mozilla, проект
  - версии JavaScript, 290
  - загрузка Rhino, 315
- multiline, свойство объекта RegExp, 288, 858
- multiple, свойство
  - Input, объект, 965
  - Select, элемент, 1007
- muted, свойство
  - MediaElement, объект, 989
  - управление воспроизведением, 660

N

- \\n (символ перевода строки), 59
- name, атрибут HTML-элементов, 381, 431

  выбор элементов документа по значению атрибута name, 394
  создание свойств в объекте Document, 395
name, свойство
  Attr, объект, 886
  DocumentFragment, объект, 922
  DOMException, объект, 923
  Error, объект, 804, 805
  EvalError, объект, 809
  File, объект, 735, 946
  FormControl, объект, 952
  Form, объект, 950
  IFrame, объект, 961
  ReferenceError, объект, 857
  URIError, объект, 880
  Window, объект, 383, 1024
  элементов форм, 432
namedItem(), метод
  HTMLCollection, объект, 397, 959
  HTMLOptionsCollection, объект, 961
  Select, элемент, 1009
namespaceURI, свойство объекта Attr, 886
namespaceURI, свойство объекта Element, 927
NaN, константа (объект Number), 833
NaN (нечисло), 54, 830
  isNaN(), функция, 817
  Number.NaN, константа, 833
  и операции сравнения, 54
naturalHeight и naturalWidth, свойства объекта Image, 963
Navigator, объект, 359, 374, 996
  appName, свойство, 375
  appVersion, свойство, 375
  cookieEnabled, свойство, 635
  cookiesEnabled(), метод, 376
  geolocation, свойство, 376, 708
  javaEnabled(), метод, 376
  online, свойство, 650
  onLine, свойство, 376
  platform, свойство, 375
  userAgent, свойство, 375
  свойства, определяющие тип броузера, 374
navigator, свойство
  Window, объект, 374, 1024
  WorkerGlobalScope, объект, 724, 1031
NEGATIVE_INFINITY, константа (объект Number), 833
NETWORK, раздел в файле объявления кэшируемого приложения, 645
networkState, свойство объекта MediaElement, 662, 989
new, ключевое слово
  в выражениях создания объектов, 84
  создание экземпляров классов языка Java, 316
new, оператор
  вызов конструктора, 224
  создание объектов, 140
newURL, свойство объекта HashChangeEvent, 957
newValue, свойство объекта StorageEvent, 1010
next() и prev(), методы объекта jQuery, 620
next(), метод
  генераторов, 304
  итераторов, 301
nextAll()и prevAll(), методы объекта jQuery, 620
nextElementSibling, свойство объекта Element, 402, 927
nextSibling, свойство объекта Node, 401, 998
nextUntil() и prevUntil(), методы объекта jQuery, 621
noConflict(), функция, 560, 623
Node, интерпретатор, 314
  электронная документация, 322
Node, объект, 997
  appendChild(), метод, 413
  attributes, свойство, 408
  cloneNode(), метод, 413
  insertBefore(), метод, 413
  removeChild(), метод, 415
  replaceChild(), метод, 415
  textContent, свойство, 410
  документы как деревья узлов, 401
  константы, определяющие возможные значения, возвращаемые методом compareDocumentPosition(), 998
  константы, определяющие возможные значения свойства nodeType, 998
  методы, 999
  свойства, 998
NodeList, объект, 394, 919, 1001
  возвращается методом getElementsByTagName(), 395
  обзор, 396
nodeName, свойство объекта Node, 401, 998
nodeType, свойство объекта Node, 401, 999
nodeValue, свойство объекта Node, 401, 412, 999
normalize(), метод объекта Node, 1001
not(), метод объекта jQuery, 619
noValidate, свойство объекта Form, 950
now(), метод объекта Date, 789
null, значение, 62, 63
  и выражения обращения к свойствам, 82
  свойства, ошибка доступа, 146
Number, объект, 831
  MAX_VALUE, константа, 833
  MIN_VALUE, константа, 833
  NaN, константа, 833
  NEGATIVE_INFINITY, константа, 833
  POSITIVE_INFINITY, константа, 834
  toExponential(), метод, 834
  toFixed(), метод, 835
  toLocaleString(), метод, 836
  toPrecision(), метод, 836
  toString(), метод, 69, 837
  valueOf(), метод, 838
  константы, 831
  методы, 831
Number(), конструктор, 64, 831
Number(), функция, преобразование типов, 69

O

<object>, элемент, 571, 658
  отображение изображений в формате SVG, 666
Object, класс, 97, 838
  constructor, свойство, 838, 840
  create(), функция, 156, 157, 841
  defineProperties(), функция, 156, 842
  defineProperty(), функция, 124, 155, 158, 168, 842
  freeze(), функция, 161, 843
  getOwnPropertyDescriptor(), функция, 158, 844
  getOwnPropertyNames(), функция, 151, 845
  getPrototypeOf(), функция, 158, 845

- hasOwnProperty(), метод, 148, 846
- isExtensible(), функция, 160, 847
- isFrozen(), функция, 161, 847
- isPrototypeOf(), метод, 159, 848
- isSealed(), функция, 161, 848
- keys(), функция, 151, 849
- preventExtensions(), функция, 160, 850
- propertyDescriptor(), функция, 155
- propertyIsEnumerable(), метод, 148, 850
- prototype, свойство, 140
- seal(), функция, 160, 851
- toLocaleString(), метод, 851
- toString(), метод, 852
- valueOf(), метод, 853
- методы, 162, 838
  - статические, 839

Object(), функция, преобразование типов, 69
Offline Web Applications API, 628
offset, свойства элементов документа, 426
offset(), метод объекта jQuery, 568
offsetHeight и offsetWidth, свойства объекта Element, 426, 927
offsetLeft и offsetTop, свойства объекта Element, 426, 927
offsetParent, свойство объекта Element, 426, 927
offsetParent(), метод объекта jQuery, 569
offsetX и offsetY, свойства объекта Event, 939
oldURL, свойство объекта HashChangeEvent, 957
oldValue, свойство объекта StorageEvent, 1010
onbeforeunload, обработчик события окна, 495
onchange, атрибут, 341
onchange, обработчик событий
- переключателей и флажков в формах, 434
- текстовых полей ввода, 435

onclick, обработчик событий, 343
- кнопки в формах, 433

onclose, свойство объекта WorkerGlobalScope, 722
ondragstart, обработчик событий, 510
one(), метод объекта jQuery, 580
onerror, свойство
- Window, объект, 379, 482
  - присваивание функции, 349
- WorkerGlobalScope, объект, 724

onhashchange, свойство объекта Window, 711
onLine, свойство
- Navigator, объект, 376, 996
- WorkerNavigator, объект, 1033

onload, обработчик событий, 345
- в программе часов с цифровым табло, 338

onLoad(), функция (пример), 349
onmessage, свойство объекта WorkerGlobalScope, 722
onmousedown, атрибут элемента <div>, 504
onreadystatechange, свойство объекта XMLHttpRequest, 532, 1039
onreset, обработчик событий элементов форм, 432
onstorage, свойство объекта Window, 633
onsubmit, обработчик событий элементов форм, 432
ontimeout, свойство объекта XMLHttpRequest, 544
opacity, свойство, 459, 586
- анимационные эффекты, 588

open(), метод
- Document, объект, 920
- Window, объект, 383, 1027
- XMLHttpRequest, объект, 1037

opener, свойство объекта Window, 385, 1024
OpenSocial API, 294
Opera, веб-броузер
- глобальный подход к композиции, 700
- текущая версия, 355

optimum, свойство объекта Meter, 995
Option, объект, 1002
Option, элемент, 435
options, свойство элемента Select, 1008
orientation, свойство объекта Window, 489
orientationchanged, событие, 489
origin, свойство объекта MessageEvent, 717, 993
originalEvent, свойство объекта Event, 579
outerHeight и outerWidth, свойства объекта Window, 1024
outerHTML, свойство объекта Element, 409, 928
outerWidth() и outerHeight(), методы объекта jQuery, 569
Output, объект, 1003
overflow, свойство, 459, 483
overrideMimeType(), метод объекта XMLHttpRequest, 535, 1037
ownerDocument, свойство объекта Node, 999
ownerNode, свойство объекта CSSStyleSheet, 911
ownerRule, свойство объекта CSSStyleSheet, 911

P

padding, свойство, 447
PageTransitionEvent, объект, 1003
pageX и pageY, свойства объекта Event, 939
- в библиотеке jQuery, 578

pageXOffset и pageYOffset, свойства объекта Window, 422, 1024
param(), функция, 600
parent, свойство объекта Window, 1024
parent() и parents(), методы объекта jQuery, 621
parentNode, свойство объекта Node, 401, 999
parentRule, свойство объекта CSSRule, 909
parentStyleSheet, свойство
- CSSRule, объект, 909
- CSSStyleSheet, объект, 911

parentsUntil(), метод объекта jQuery, 621
parse(), метод объекта Date, 790
parse(), функция, 161, 819
parseFloat(), функция, 70, 854
parseInt(), функция, 70, 855
parseJSON(), функция, 612
pathname, свойство
- Link, объект, 985
- Location, объект, 371, 987
- WorkerLocation, объект, 1033

path, атрибут cookies, 636
pattern, переменная, 277
pattern, свойство объекта Input, 965
patternMismatch, свойство объекта FormValidity, 954
paused, свойство объекта MediaElement, 989
pause(), метод объекта MediaElement, 660, 992
persisted, свойство объекта PageTransitionEvent, 1003
pixelDepth, свойство объекта Screen, 1006
PI, константа (объект Math), 828
placeholder, атрибут текстовых полей ввода, 435
placeholder, свойство объекта Input, 966
platform, свойство
- Navigator, объект, 375, 996
- WorkerNavigator, объект, 1033

play(), метод объекта MediaElement, 660, 992
playbackRate, свойство объекта MediaElement, 660, 989
played, свойство объекта MediaElement, 661, 989
plugins, свойство
  Document, объект, 917
  HTMLDocument, объект, 396
PNG, формат изображений, получение содержимого холста, 698
pop(), метод объекта Array, 169, 175, 771
popstate, событие, 712, 713
PopStateEvent, объект, 1004
port1 и port2, свойства объекта MessageChannel, 993
port, свойство
  Link, объект, 985
  Location, объект, 371, 987
  WorkerLocation, объект, 1033
ports, свойство объекта MessageEvent, 994
position, свойство, 451
  объекта Progress, 1005
position(), метод объекта jQuery, 569
POSITIVE_INFINITY, константа (объект Number), 834
POSIX (UNIX) API, поддержка в Node, 325, 326
POST, метод, 529
  выгрузка файлов посредством HTTP-запросов, 540
  выполнение HTTP-запросов с данными
    в формате HTML-форм, 537
    в формате JSON, 538
    в формате XML, 538
  передача простого текста серверу, 531
  тело запроса, 531
post(), функция, 600
poster, свойство объекта Video, 1020
postMessage(), метод
  MessagePort, объект, 994
  Window, объект, 716, 1027
    модуль поиска на сайте Twitter, 718
  Worker, объект, 721, 1030
  WorkerGlobalScope, объект, 1032
pow(), функция, объект Math, 828
prefix, свойство
  Attr, объект, 886
  Element, объект, 928
preload, свойство объекта MediaElement, 660, 989
prepend(), метод объекта jQuery, 572
prependTo(), метод объекта jQuery, 572
prev(), метод объекта jQuery, 620
prevAll(), метод объекта jQuery, 620
preventDefault(), метод объекта Event, 497, 584, 941
preventExtensions(), функция, 160, 850
previousElementSibling, свойство объекта Element, 402, 928
previousSibling, свойство объекта Node, 401, 999
prevUntil(), метод объекта jQuery, 621
print(), метод объекта Window, 1027
print(), функция (Rhino), 316
processData, параметр, 600
ProcessingInstruction, объект, 1004
profile(), метод
  Console, объект, 906
  ConsoleCommandLine, объект, 908
profileEnd(), метод
  Console, объект, 907
  ConsoleCommandLine, объект, 908
Progress, объект, 1004
progress, событие, 488
ProgressEvent, объект, 1005
prompt(), метод объекта Window, 377, 1027
propertyDescriptor(), функция, 155
propertyIsEnumerable(), метод, 148, 850
protocol, свойство
  Link, объект, 985
  Location, объект, 371, 987
  WebSocket, объект, 757, 1021
  WorkerLocation, объект, 1033
prototype, атрибут, 158, 159
Prototype, библиотека, 368
prototype, свойство
  Function, объект, 814
  наследование, 140
  ограничение на использование в безопасных подмножествах, 293
  функций, 209
  функции-конструктора, 226
Prototype, фреймворк
  Scriptaculous, библиотека, 467
proxy(), функция, 612
publicId, свойство объекта DocumentFragment, 922
push(), метод объекта Array, 175, 771
  добавление элементов в конец массива, 169
pushStack(), метод объекта jQuery, 622
pushState(), метод объекта History, 712, 958
  управление историей посещений (пример), 713
putImageData(), метод объекта CanvasRenderingContext2D, 901

Q

quadraticCurveTo(), метод объекта CanvasRenderingContext2D, 685, 902
queryCommandEnabled(), метод объекта Document, 442, 920
queryCommandIndeterm(), метод объекта Document, 443, 920
queryCommandState(), метод объекта Document, 442, 920
queryCommandSupported(), метод объекта Document, 920
queryCommandValue(), метод объекта Document, 442, 920
querySelector(), метод
  Document, объект, 399, 920
  DocumentFragment, объект, 399
  Element, объект, 930
querySelectorAll(), метод
  Document, объект, 399, 473, 920
    в сравнении с функцией $(), 564
  DocumentFragment, объект, 399
  Element, объект, 930
  выбор элементов форм, 430
queue, свойство объекта с параметрами анимационного эффекта, 591
queue(), метод объекта jQuery, 594

R

random(), функция, объект Math, 829
Range, объект, 440
RangeError, объект, 856
rangeOverflow, свойство объекта FormValidity, 954

rangeUnderflow, свойство объекта FormValidity, 954
readAsArrayBuffer(), метод
  FileReader, объект, 740, 741, 948
  FileReaderSync, объект, 949
readAsBinaryString(), метод
  FileReader, объект, 740, 948
  FileReaderSync, объект, 949
readAsDataURL(), метод
  FileReader, объект, 740, 948
  FileReaderSync, объект, 949
readAsText(), метод
  FileReader, объект, 740, 741, 948
  FileReaderSync, объект, 949
readOnly, свойство объекта Input, 966
readyState, свойство
  Document, объект, 351, 498, 917
  EventSource, объект, 943
  FileReader, объект, 740, 947
  MediaElement, объект, 662, 990
  WebSocket, объект, 1021
  XMLHttpRequest, объект, 532, 1035
    значения, 532
readystatechange, событие, 498, 532
rect(), метод объекта CanvasRenderingContext2D, 687, 902
reduce(), метод объекта Array, 178, 215, 772
reduceRight(), метод объекта Array, 178, 773
ReferenceError, объект, 80, 856
referrer, свойство объекта Document, 437, 917
RegExp, класс, 50
  toString(), метод, 72
RegExp, объект, 60, 276, 287, 857
  exec(), метод, 858
  global, свойство, 859
  ignoreCase, свойство, 860
  lastIndex, свойство, 860
  source, свойство, 860
  test(), метод, 861
  toString(), метод, 861
  как вызываемый объект, 214
  методы поиска по шаблону, 288
  свойства экземпляра, 858
RegExp(), конструктор, 277, 287
registerContentHandler(), метод объекта Navigator, 997
registerProtocolHandler(), метод объекта Navigator, 997
relatedTarget, свойство объекта Event, 484, 578, 939
relList, свойство объекта Link, 985
reload(), метод объекта Location, 373, 987
remove(), метод
  DOMTokenList, объект, 470, 925
  jQuery, объект, 574
  HTMLOptionsCollection, объект, 961
  Select, элемент, 1009
removeAttr(), функция, 565
removeAttribute(), метод объекта Element, 407, 930
removeAttributeNS(), метод объекта Element, 930
removeChild(), метод объекта Node, 415, 1001
removeClass(), метод объекта jQuery, 567
removeData(), метод объекта jQuery, 571
removeEventListener(), метод
  Document, объект, 492
  EventTarget, объект, 945
  WorkerGlobalScope, объект, 724
  Worker, объект, 722
removeItem(), метод объекта Storage, 1010
repeat, свойство объекта Event, 942
replace(), метод
  Location, объект, 372, 987
  String, объект, 285, 869
replaceAll(), метод объекта jQuery, 572
replaceChild(), метод объекта Node, 415, 1001
replaceData(), метод
  Comment, узел, 905
  Text, объект, 1015
replaceState(), метод объекта History, 713, 959
replaceWith(), метод объекта jQuery, 572
required, свойство объекта Input, 966
reset(), метод объекта Form, 433, 951
resize, событие окон, 483
resize(), метод объекта jQuery, 575
response, свойство объекта XMLHttpRequest, 1035
responseText, свойство объекта XMLHttpRequest, 535, 1035
  декодирование ответа, 534
responseType, свойство объекта XMLHttpRequest, 1035
responseXML, свойство объекта XMLHttpRequest, 534, 535, 1036
restore(), метод объекта CanvasRenderingContext2D, 691, 902
result, свойство
  Event, объект, 577, 579
  FileReader, объект, 740, 947
return, инструкция, 83, 127, 188
  использование в конструкторах, 192
  использование в функциях-генераторах, 304
returnValue, свойство
  BeforeUnloadEvent, объект, 886
  Event, объект, 497
  Window, объект, 1024
reverse(), метод объекта Array, 172, 774
revokeObjectURL(), метод объекта URL, 739, 1019
RGBA, формат определения цвета, 458
Rhino, интерпретатор, 314
  поддержка расширений JavaScript, 290
  поддержка расширения E4X (ECMAScript for XML), 310
  создание графического интерфейса (пример), 319
right, свойство объекта ClientRect, 904
right и bottom, свойства стиля, 456, 457
rotate(), метод объекта CanvasRenderingContext2D, 681, 902
rotation, свойство, 488
round(), функция, объект Math, 829
rows, свойство
  Table, объект, 1012
  TableSection, объект, 1014
rowIndex, свойство объекта TableRow, 1013
rowSpan, свойство объекта TableCell, 1013
RSH (Really Simple History – действительно простое управление историей), библиотека, 374
rules, свойство вместо cssRules, 473

S

\\s, метасимвол в регулярных выражениях, 279
\\S, непробельный символ Unicode, 279
Safari, веб-броузер, 504

  textInput, событие, 486
  текущая версия, 355
sandbox, свойство объекта IFrame, 961
save(), метод объекта CanvasRenderingContext2D, 691, 902
scale(), метод объекта CanvasRenderingContext2D, 681, 902
scale, свойство, 488
scoped, свойство объекта Style, 1011
Screen, объект, 377, 1006
  availHeight, свойство, 377
  availWidth, свойство, 377
  height, свойство, 377
  width, свойство, 377
screen, свойство объекта Window, 377, 1024
screenX и screenY, свойства
  Event, объект, 940
  Window, объект, 1024
<script>, элемент, 335
  defer и async, атрибуты, 346
  src, атрибут, 339
  type, атрибут, 341
  встраивание JavaScript в HTML, 23, 338
  как транспорт в архитектуре Ajax, 525
  работа с протоколом HTTP, 535
  текстовое содержимое, 411
  удаление угловых скобок для предотвращения нападений по методике межсайтового скриптинга, 366
Script, объект, 1007
Scriptaculous, библиотека из фреймворка Prototype, 368, 467
scripts, свойство объекта Document, 396, 917
scroll, свойства элементов документа, 426
scroll(), метод
  jQuery, объект, 575
  Window, объект, 425, 1028
scrollBy(), метод объекта Window, 425, 1028
scrollHeight и scrollWidth, свойства объекта Element, 928
scrollIntoView(), метод объекта Element, 931
scrollLeft и scrollTop, свойства объекта Element, 422, 928
scrollLeft(), метод объекта jQuery, 570
scrollTo(), метод объекта Window, 425, 1028
scrollTop(), метод объекта jQuery, 570
seal(), функция, 160, 851
seamless, свойство объекта IFrame, 962
search, свойство
  Link, объект, 985
  Location, объект, 371, 987
  WorkerLocation, объект, 1033
search(), метод объекта String, 285, 871
sectionRowIndex, свойство объекта TableRow, 1013
secure, атрибут cookies, 637
seekable, свойство объекта MediaElement, 661, 990
seeking, свойство объекта MediaElement, 990
Select, объект, 1007
Select, элемент, 435
select(), метод
  Input, объект, 966
  jQuery, объект, 575
  TextArea, объект, 1016
selected, свойство объекта Option, 436, 1002
selectedIndex, свойство
  HTMLOptionsCollection, объект, 960
  Select, элемент, 436, 1008
selectedOption, свойство объекта Input, 966
selectedOptions, свойство объекта Select, 1008
Selection, объект, 440
selectionEnd, свойство
  Input, объект, 966
  TextArea, объект, 1016
selectionStart, свойство
  Input, объект, 966
  TextArea, объект, 1016
selector, свойство объекта jQuery, 562
selectorText, свойство объекта CSSRule, 909
self, переменная, использование во вложенных функциях, 192
self, свойство
  Window, объект, 1025
  WorkerGlobalScope, объект, 1031
send(), метод
  WebSocket, объект, 1021
  XMLHttpRequest, объект, 530, 1037
  генераторов, 306
serialize(), метод, 599
Server-Sent Events, стандарт, 550
  имитация объекта EventSource с помощью объекта XMLHttpRequest, 552
  простой клиент чата на основе объекта EventSource, 551
  сервер чата, 554
sessionStorage, свойство объекта Window, 630, 1025
  прикладной программный интерфейс объекта Storage, 632
  события, 633
  срок хранения и область видимости, 630
set(), метод типизированных массивов, 729, 1019
setAttribute(), метод объекта Element, 406, 931
setCapture(), метод (IE), 501
setData(), метод объекта DataTransfer, 509, 913
setDate(), метод объекта Date, 790
setDragImage(), метод объекта DataTransfer, 509, 913
setFloat32(), метод объекта DataView, 914
setFloat64(), метод объекта DataView, 914
setFullYear(), метод объекта Date, 791
setHours(), метод объекта Date, 791
setInt8(), метод объекта DataView, 914
setInt16(), метод объекта DataView, 914
setInt32(), метод объекта DataView, 914
setInterval(), метод
  Window, объект, 370, 1028
  WorkerGlobalScope, объект, 1032
setInterval(), функция, 323, 349
  использование в злонамеренном коде, 367
setItem(), метод объекта Storage, 1010
setMilliseconds(), метод объекта Date, 792
setMinutes(), метод объекта Date, 792
setMonth(), метод объекта Date, 792
setRequestHeader(), метод объекта XMLHttpRequest, 530, 1038
setSeconds(), метод объекта Date, 793
setSelectionRange(), метод
  Input, объект, 966
  TextArea, объект, 1016
setTime(), метод объекта Date, 793
setTimeout(), метод
  Window, объект, 334, 370, 1028

  WorkerGlobalScope, объект, 1032
setTimeout(), функция, 323, 349
  ограничение времени ожидания выполнения запроса объектом XMLHttpRequest, 544
setTransform(), метод объекта CanvasRenderingContext2D, 681, 683, 903
setUint8(), метод объекта DataView, 914
setUint16(), метод объекта DataView, 914
setUint32(), метод объекта DataView, 914
setUTCDate(), метод объекта Date, 793
setUTCFullYear(), метод объекта Date, 794
setUTCHours(), метод объекта Date, 794
setUTCMilliseconds(), метод объекта Date, 795
setUTCMinutes(), метод объекта Date, 795
setUTCMonth(), метод объекта Date, 795
setUTCSeconds(), метод объекта Date, 796
setVersion(), метод объекта IndexedDB, 750
setYear(), метод объекта Date, 796
shadowBlur, свойство, 695, 896
shadowColor, свойство, 695, 896
shadowOffsetX и shadowOffsetY, свойства, 695
  CanvasRenderingContext2D, объект, 896
sheet, свойство объекта Link, 985
sheet, свойство объекта Style, 1011
shift(), метод объекта Array, 169, 175, 774
shiftKey, свойство, 500, 519
  Event, объект, 940
  событий мыши, 483
show(), метод объекта jQuery, 588, 591
showModalDialog(), метод объекта Window, 378, 1028
sibling, свойства объекта Element, 402
siblings(), метод объекта jQuery, 620
sin(), функция, объект Math, 829
size, свойство
  Blob, объект, 887
  Input, объект, 966
slice(), метод
  Array, объект, 174, 775
  Blob, объект, 732, 887
  jQuery, объект, 618
  String, объект, 871
slideDown(), slideUp() и slideToggle(), методы объекта jQuery, 589
some(), метод объекта Array, 178, 775
sort(), метод объекта Array, 173, 776
  функции как аргументы, 200
<source>, элемент, 658
source, свойство
  MessageEvent, объект, 717, 994
  RegExp, объект, 287, 858, 860
sparkline, встроенные диаграммы, 408, 704
speed, свойство объекта Geocoordinates, 955
Spidermonkey, интерпретатор
  поддержка расширений JavaScript, 290
  поддержка расширения E4X (ECMAScript for XML), 310
  присваивание с разложением, 298
splice(), метод объекта Array, 174, 777
split(), метод
  String, объект, 287, 872
  разбиение содержимого cookie на пары имя/значение при его анализе, 638
splitText(), метод объекта Text, 1015
SQRT1_2, константа (объект Math), 830
SQRT2, константа (объект Math), 830
sqrt(), функция, объект Math, 829
src, атрибут элемента <script>, 339
src, свойство
  IFrame, объект, 962
  Image, объект, 963
  MediaElement, объект, 990
  Script, объект, 1007
srcdoc, свойство объекта IFrame, 962
srcElement, свойство объекта Event, 940
start(), метод
  MessagePort, объект, 994
  TimeRanges, объект, 1017
startOffsetTime, свойство объекта MediaElement, 990
state, свойство
  History, объект, 713
  PopStateEvent, объект, 1004
status, свойство
  ApplicationCache, объект, 649
  XMLHttpRequest, объект, 1036
statusText, свойство объекта XMLHttpRequest, 1036
step, свойство объекта Input, 966
stepDown(), метод объекта Input, 967
stepMismatch, свойство объекта FormValidity, 954
stepUp(), метод объекта Input, 967
stop(), метод объекта jQuery, 593
stopImmediatePropagation(), метод объекта Event, 498, 941
StopIteration, исключение, 301
  возбуждение методом next() генераторов, 304
stopPropagation(), метод объекта Event, 497, 941
Storage, объект, 1009
storageArea, свойство объекта StorageEvent, 1010
StorageEvent, объект, 1010
String, объект, 862
  charAt(), метод, 864
  charCodeAt(), метод, 864
  concat(), метод, 865
  fromCharCode(), метод, 515, 865
  HTML-методы, 863
  indexOf(), метод, 866
  lastIndexOf(), метод, 866
  length, свойство, 867
  localeCompare(), метод, 96, 867
  match(), метод, 868
  replace(), метод, 869
  search(), метод, 871
  slice(), метод, 871
  split(), метод, 872
  substr(), метод, 874
  substring(), метод, 874
  toLocaleLowerCase(), метод, 875
  toLocaleUpperCase(), метод, 875
  toLowerCase(), метод, 96, 876
  toString(), метод, 876
  toUpperCase(), метод, 96, 876
  trim(), метод, 876
  valueOf(), метод, 877
  методы, 862
    статические, 863
String(), конструктор, 64
String(), функция, преобразование типов, 69
stringify(), функция, 161, 820
stroke(), метод объекта CanvasRenderingContext2D, 675, 691, 903

strokeRect(), метод объекта CanvasRenderingContext2D, 687, 903
strokeStyle, свойство объекта CanvasRenderingContext2D, 678, 896
strokeText(), метод объекта CanvasRenderingContext2D, 692, 903
<style>, элемент, включение таблиц стилей, 446
style, атрибут, 31
Style, объект, 1011
style, свойство, 334
  CSSRule, объект, 909
  Element, объект, 334, 928
styleSheets, свойство объекта Document, 473, 917
subarray(), метод типизированных массивов, 730, 1019
submit(), метод
  Form, объект, 432, 951
  jQuery, объект, 575
    возбуждение событий, 582
submit() и reset(), методы объекта Form, 432
substr(), метод объекта String, 874
substring(), метод объекта String, 874
substringData(), метод
  Comment, узел, 905
  Text, объект, 1015
support, свойство, 613
<svg:path>, элемент, 669
SVG (Scalable Vector Graphics – масштабируемая векторная графика), формат, 665
  круговая диаграмма в формате SVG, построенная JavaScript-сценарием, 667
  отображение времени посредством манипулирования SVG-изображением, 670
swapCache(), метод объекта ApplicationCache, 649, 883
switch, инструкция, 117
  case, конструкция, 117
SyntaxError, объект, 877
  ошибка при удалении свойств в строгом режиме, 148
systemId, свойство объекта DocumentFragment, 922

T
Table, объект, 1011
TableCell, объект, 1013
TableRow, объект, 1013
TableSection, объект, 1014
tagName, свойство объекта Element, 928
tan(), функция, объект Math, 830
target, свойство
  Event, объект, 940
    в библиотеке jQuery, 578
  Form, объект, 431, 951
  ProcessingInstruction, объект, 1004
  событий, 477
tBodies, свойство объекта Table, 1012
terminate(), метод объекта Worker, 722
test(), метод объекта RegExp, 289, 858, 861
Text, объект, 1014
Text и Textarea, элементы форм, 432
text, свойство
  Link, объект, 986
  Option, объект, 436, 1003
  Script, объект, 1007
Text, узлы, 402
  как содержимое элементов, 411
  создание, 413
text(), метод объекта jQuery, 568
text-shadow, свойство, 454
textAlign, свойство объекта CanvasRenderingContext2D, 692, 896
TextArea, объект, 1015
textBaseline, свойство объекта CanvasRenderingContext2D, 692, 896
textContent, свойство объекта Node, 410, 999
textLength, свойство объекта TextArea, 1016
TextMetrics, объект, 693, 901, 1016
TextRange, объект (IE), 440
tFoot, свойство объекта Table, 1012
tHead, свойство объекта Table, 1012
this, ключевое слово
  в вызовах методов, 191
  в обработчиках событий, 433
  в функциях, используемых как методы, 191
  использование методов чтения и записи свойств, 153
  как первичное выражение, 80
  контекст вызова функции, 185
  ограничение на использование в безопасных подмножествах, 292
  ссылка на глобальный объект, 64
  ссылка на целевой объект в обработчиках событий, 493
throw, инструкция, 128
throw(), метод генераторов, 306
time(), метод объекта Console, 907
timeEnd(), метод объекта Console, 907
timeout, параметр объекта Geolocation, 956
timeout, свойство объекта XMLHttpRequest, 544, 1036
TimeRanges, объект, 661, 1016
timeStamp, свойство объекта Event, 578, 940
timestamp, свойство объекта Geoposition, 957
title, свойство
  CSSStyleSheet, объект, 472, 911
  Document, объект, 437, 917
  Element, объект, 928
  Link, объект, 986
  Style, объект, 1011
toArray(), метод объекта jQuery, 562
toDataURL(), метод объекта Canvas, 698, 889
toDateString(), метод объекта Date, 796
toElement, свойство объекта Event, 940
toExponential(), метод объекта Number, 70, 834
toFixed(), метод объекта Number, 70, 835
toggle(), метод
  DOMTokenList, объект, 470, 925
  jQuery, объект, 576, 588
toggleClass(), метод объекта jQuery, 567
toGMTString(), метод объекта Date, 797
toISOString(), метод объекта Date, 797
toJSON(), метод, 163
  Date, объект, 797
  реализация в классах, 243
toLocaleDateString(), метод объекта Date, 798
toLocaleLowerCase(), метод объекта String, 875
toLocaleString(), метод, 163
  Array, объект, 176, 778
  Date, объект, 798
  Number, объект, 836
  Object, класс, 851

  реализация в классах, 243
toLocaleTimeString(), метод объекта Date, 799
toLocaleUpperCase(), метод объекта String, 875
toLowerCase(), метод объекта String, 876
tooLong, свойство объекта FormValidity, 954
top, left, width и height, свойства стиля, 456, 457
top, свойство
  ClientRect, объект, 904
  Window, объект, 386, 1025
toPrecision(), метод объекта Number, 70, 836
toString(), метод, 162
  Array, объект, 176, 778
  Boolean, объект, 780
  Date, объект, 799
  Error, объект, 804, 806
  Function, объект, 814
  Location, объект, 371
  Number, объект, 69, 837
  Object, класс, 852
  RegExp, объект, 861
  Selection, объект, 440
  String, объект, 876
  определение класса объекта, 159
  преобразование логических значений в строки, 62
  преобразование типов, 71
  реализация в классах, 242
  функций, 213
total, свойство объекта ProgressEvent, 542, 1006
toTimeString(), метод объекта Date, 799
toUpperCase(), метод объекта String, 876
toUTCString(), метод объекта Date, 800
trace(), метод объекта Console, 907
transform(), метод объекта CanvasRenderingContext2D, 683, 903
translate(), метод объекта CanvasRenderingContext2D, 681
trigger(), метод объекта jQuery, 577, 582
triggerHandler(), метод объекта jQuery, 584
trim(), метод объекта String, 876
trim(), функция, 613
try/catch/finally, инструкция, 129
  множественные блоки catch, 309
Twitter, сайт, модуль поиска с помощью метода postMessage(), 718
type, атрибут элемента <script>, 341
type, свойство
  Blob, объект, 887
  CSSRule, объект, 909
  CSSStyleSheet, объект, 911
  Event, объект, 940
  FormControl, объект, 952
  Script, объект, 1007
  Select, элемент, 436
  Style, объект, 1011
  событий, 477
  элементов форм, 429, 430, 432
TypeError, объект, 64, 878
  возбуждение ошибки при преобразовании типов, 69
  как результат попытки удалить свойство, 147
  ошибка при попытке создать или изменить свойства, 156
  ошибки доступа к свойствам, 146
typeMismatch, свойство объекта FormValidity, 954
typeof, оператор, 85, 105, 232
  и объекты XML, 310
  применение к значениям null и undefined, 62
types, свойство объекта DataTransfer, 511, 912

U
Uint8Array, класс, 728, 1017
Uint16Array, класс, 728, 1017
Uint32Array, класс, 728, 1017
unbind(), метод объекта jQuery, 581
undefined, значение, 62, 63
  возвращаемое функцией, 188
  и выражения обращения к свойствам, 82
  свойства, ошибка доступа, 146
undelegate(), метод объекта jQuery, 585
unescape(), функция, 879
unload, событие, 482
  BeforeUnloadEvent, объект, 886
unmonitorEvents(), метод объекта ConsoleCommandLine, 908
unshift(), метод объекта Array, 175, 176, 779
unwrap(), метод объекта jQuery, 574
update(), метод объекта ApplicationCache, 649, 884
upload, свойство объекта XMLHttpRequest, 542, 1036
URIError, объект, 879
URI (Uniform Resource Identifier – унифицированный идентификатор ресурса), 801
  decodeURI(), функция, 801
  decodeURIComponent(), функция, 802
  encodeURI(), функция, 802
  encodeURIComponent(), функция, 803
URL-адреса
  URL, объект, 1019
  анализ, 371
  аргументы функции importScripts(), 722
  веб-сокетов, 756
  двоичных объектов, 739
  относительные, 373
  шаблонные, в объявлениях кэшируемых приложений, 645
URL, объект, 1019
url, свойство
  EventSource, объект, 943
  StorageEvent, объект, 1011
  WebSocket, объект, 1021
URL, свойство
  Document, объект, 437, 917
  Window, объект, 1025
userAgent, свойство
  Navigator, объект, 375, 996
  WorkerNavigator, объект, 1034
userData API, 628
use strict, директива, 133
UTC (Universal Coordinated Time – универсальное скоординированное время), 781
UTC(), метод объекта Date, 800

V
val(), метод объекта jQuery, 567
valid, свойство объекта FormValidity, 954
validationMessage, свойство объекта FormControl, 952
validity, свойство объекта FormControl, 952
value, атрибут (свойств), 154
value, свойство
  DOMSettableTokenList, объект, 924

  FileUpload, объект, ограничение возможностей из соображений безопасности, 362
  FormControl, объект, 952
  Meter, объект, 995
  Option, объект, 1003
  Progress, объект, 1005
  элементов форм, 431, 432
valueAsDate, свойство объекта Input, 966
valueAsNumber, свойство объекта Input, 966
valueMissing, свойство объекта FormValidity, 954
valueOf(), метод, 163
  Boolean, объект, 780
  Date, объект, 801
  Number, объект, 838
  Object, класс, 853
  String, объект, 877
  преобразование типов, 72
  реализация в классах, 243
values(), метод объекта ConsoleCommandLine, 908
var, инструкция, 112
var, ключевое слово, 74
<video>, элемент, 658
  controls, атрибут, 658
  события, 486
Video, объект, 1019
videoHeight и videoWidth, свойства объекта Video, 1020
view, свойство объекта Event, 940
void, оператор, 85, 108
  принудительная подстановка значения undefined, возвращаемого выражением, 343
volume, свойство
  MediaElement, объект, 990
  управление воспроизведением, 660

W

\\W (неслово, метасимвол), 279, 283
\\w (слово, метасимвол), 279, 283
W3C, консорциум
  стандарт XMLHttpRequest Level 2 (XHR2), 527
WAI-ARIA, стандарт, 361
warn(), метод объекта Console, 907
wasClean, свойство объекта CloseEvent, 904
watchPosition(), метод объекта Geolocation, 708, 956
WebGL, прикладной интерфейс, 673
webkitURL.revokeObjectURL(), метод, 739
WebSocket, объект, 1020
WebSocket API, спецификация, 755
Web Storage API, 627
  спецификация, 488
Web Workers, спецификация, 720
  Worker, объект, 721
«Web Workers», фоновые потоки выполнения, 349
wheelDelta, свойства, 505
wheelDelta, свойство объекта Event, 940
wheelDeltaX и wheelDeltaY, свойства объекта Event, 940
which, свойство объекта Event, 483, 578, 941
while, циклы, 119
  continue, инструкция, 127
wholeText, свойство объекта Text, 1014
width, свойство
  ClientRect, объект, 904
  IFrame, объект, 962
  ImageData, объект, 964
  Image, объект, 963
  Screen, объект, 377
  TextMetrics, объект, 1016
width и height, свойства
  Canvas, объект, 888
  Screen, объект, 1006
  Video, объект, 1020
  объекта контекста холста, 679
width и height, свойства стиля, 456, 457
width(), метод объекта jQuery, 569
willValidate, свойство объекта FormControl, 953
Window, объект, 63, 1022
  alert(), метод, 377
  applicationCache, свойство, 646
  closed, свойство, 385
  confirm(), метод, 377
  dialogArguments, свойство, 378
  document, свойство, 334, 390
  DOMContentLoaded, событие, 498
  event, свойство, 493
  frames, свойство, 386
  frameElement, свойство, 386
  getComputedStyle(), метод, 468
  getSelection(), метод, 440
  history, свойство, 373
  length, свойство, 387
  load, событие, 498
  localStorage и sessionStorage, свойства, 630
  location, свойство, 371
  name, свойство, 383
  navigator, свойство, 374
  onbeforeunload, обработчик события, 495
  onerror, свойство, 349, 379, 482
  onhashchange, свойство, 711
  onload, обработчик события, 335
  open(), метод, 343, 383
  opener, свойство, 385
  postMessage(), метод, 716
    модуль поиска на сайте Twitter, 718
  prompt(), метод, 377
  rotation, свойство, 488
  scale, свойство, 488
  screen, свойство, 377, 1006
  scroll(), метод, 425
  scrollBy(), метод, 425
  scrollTo(), метод, 425
  setInterval(), метод, 370
  setTimeout(), метод, 370
  showModalDialog(), метод, 378
  storage, событие, 488
  top, свойство, 386
  URL, свойство, 1019
  конструкторы, 1025
  методы, 1025
  обработка ошибок, 379
  обработчики событий, 1028
  свойства, 1023
  события, 482
    beforeprint и afterprint, 488
    offline и online, 487
  средоточие всех особенностей и прикладных интерфейсов клиентского JavaScript, 333
window, свойство объекта Window, 334, 1025
WindowProxy, объект, 389
with, инструкция, 131

  - ограничение на использование в безопасных подмножествах, 292
- withCredentials, свойство объекта XMLHttpRequest, 546, 1036
- Worker, объект, 721, 1029
  - дополнительные особенности, 724
  - обработчики событий, 1030
- Worker(), конструктор, 725
- WorkerGlobalScope, объект, 722, 1031
  - свойства, 723
- WorkerLocation, объект, 1032
- WorkerNavigator, объект, 1033
- wrap(), метод объекта jQuery, 574
- wrapAll(), метод объекта jQuery, 574
- wrapInner(), метод объекта jQuery, 574
- writable, атрибут (свойств), 154, 155
- write(), метод объекта Document, 346, 351, 438, 921
- writeFile(), функция, 326
- writeFileSync(), функция, 326
- writeln(), метод объекта Document, 439, 921
- ws:// или wss://, протоколы, 756

X

- XHR2, стандарт, 527
  - FormData, объект, 540
  - timeout, свойство, 544
- XHTML, язык разметки
  - <script>, элемент, 338
  - встраивание SVG-графики в документы, 666
  - чувствительность к регистру, 41
- XML, необязательное использование при работе с протоколом HTTP, 527
- XML, объект, 310
- XML, формат данных
  - выполнение POST-запросов с данными в формате XML, 538
- XMLHttpRequest, объект, 337, 526, 1034
  - HTTP-запросы и ответы, 529
  - responseText, свойство, 534
  - responseXML, свойство, 534
  - выполнение запроса, 529
  - выполнение синхронных запросов в фоновых потоках выполнения, 726
  - и локальные файлы, 529
  - имитация в виде объекта jqXHR в jQuery 1.5, 603
  - имитация объекта EventSource, 552
  - использование утилитами Ajax в библиотеке jQuery, 598
  - константы, определяющие возможные значения свойства readyState, 1034
  - междоменные HTTP-запросы, 545
  - методы, 1036
  - обработчики событий, 1038
  - отправка простого текста серверу методом POST, 531
  - отправка сообщений пользователя в чат, 552
  - оформление тела HTTP-запроса, 535
  - политика общего происхождения, 363
  - получение ответа, 532
  - прерывание запросов и предельное время ожидания, 544
  - свойства, 1035
  - события, возникающие в ходе
    - выгрузки файлов, 542
    - выполнения HTTP-запроса, 541
  - создание экземпляра, 527
- XMLHttpRequest(), конструктор, 725
- XMLHttpRequestUpload, объект, 1039
- XMLList, объект, 310

Y

- yield, выражение, 306
- yield, ключевое слово, 303
- yieldForStorageUpdates(), метод объекта Navigator, 997
- YUI, фреймворк, 368

Z

- z-index, свойство, 453

А

- абсолютное позиционирование элементов, 451
- абстрактные классы, 252
  - и иерархии классов, 258
- абстрактные методы, 252
- автономные веб-приложения, 643, 650, 652
  - события, 487
- альтернативы в регулярных выражениях, 281
- альфа-значение, прозрачность, 458
- анимационные эффекты
  - в библиотеке jQuery, 978
  - воспроизведение за счет управления встроенными стилями CSS, 465
  - модули CSS Transitions и Animations, 450
  - поддержка в клиентских библиотеках, 467
  - реализация собственных анимационных эффектов, 589
  - реализация с помощью библиотеки jQuery, 586
    - отмена, задержка и постановка эффектов в очередь, 593
    - простые эффекты, 588
- апострофы, 58
  - экранирование в строковых литералах, заключенных в одинарные кавычки, 58
- аппаратно-зависимые и аппаратно-независимые события, 480
- аргументы (функций), 193, 194
  - как свойства объекта, 196
  - необязательные, 193
  - списки аргументов переменной длины, 194
- арифметические выражения, 88
  - оператор +, 89
  - поразрядные операторы, 91
  - унарные арифметические операторы, 90
- арифметические операторы, 53
- асинхронная загрузка и выполнение сценариев, 346
- асинхронный ввод/вывод
  - HTTP-сервер (пример), 327
  - XMLHttpRequest и спецификация File API, версия 2, 487
  - буферы, 324
  - в интерпретаторе Node, 321
  - модуль утилит клиента HTTP в Node, 329
  - потоки ввода/вывода, 324
  - сетевые взаимодействия, 327
  - функции для работы с файлами и файловой системой в модуле fs интерпретатора Node, 325
- асинхронный обмен сообщениями между документами с различным происхождением, 717
- ассоциативность операторов, 88
- ассоциативные массивы, 143

атаки типа «отказ в обслуживании», 367
атрибуты, 405
  HTML как свойства объектов Element, 405
  в расширении E4X, 312
  данных, использование для реализации эффекта смены изображений, 657
  доступ к нестандартным HTML-атрибутам, 406
  имена изменяемых CSS-атрибутов, объект с параметрами, 590
  как узлы типа Attr, 408
  объектов, 158, 159
    class, атрибут, 159
    extensible, атрибут, 160
    prototype, атрибут, 158
  свойств, 154, 155
    копирование, 157
    получение и изменение значений, 154
  с данными, 407
  установка обработчиков событий, 490
  чтение и запись значений CSS-атрибутов методами объекта jQuery, 566
  чтение и запись значений HTML-атрибутов методами объекта jQuery, 565
аудио и видео
  определение состояния мультимедийных элементов, 661
  работа с аудио- и видеопотоками, 658
    выбор типа и загрузка, 659
    события мультимедийных элементов, 663
    управление воспроизведением, 660
аудио- и видеозаписи
  MediaElement, суперкласс, 987
  MediaError, суперкласс, 992
  Video, объект, 1019
аффинные преобразования, 682

Б

базовый JavaScript, 25
базы данных
  на стороне клиента, 747
  поддержка баз данных в броузерах, 628
безопасность, 361
  данных в cookies, 635
  и сценарии, 549
  клиентский JavaScript, 361
    атаки типа «отказ в обслуживании», 367
    межсайтовый скриптинг, 365
    неподдерживаемые возможности, 362
    ограничение возможностей, 362
    политика общего происхождения, 363
  междоменные HTTP-запросы, 545
  объект Canvas, метод toDataURL(), 698
  при сохранении данных на стороне клиента, 629
бесконечные циклы, 122
библиотеки
  визуальных эффектов, 467
  клиентские, поддержка визуальных эффектов, 467
  обеспечения совместимости, 356
  управление историей, 374
блоки инструкций, 110
блокирующее выполнение сценария, 346
блочная модель (CSS), 455
  модель border-box и свойство box-sizing, 457
  jQuery.support.boxModel, свойство, 613
букмарклеты, 343
  для работы с выделенным текстом, 440
  использование URL-адресов JavaScript, 343
буксировка (drag-and-drop), 508
  DataTransfer, объект, 911
  drag(), функция, вызов из обработчика события mousedown, 501
  буксировка элементов документа, 501
  доступ к файлам, сбрасываемым пользователем на элемент, 539
  получение файлов средствами DnD API и их выгрузка с помощью HTTP-запросов, 542
  приемник буксируемых данных, 511
  собственный источник данных, 510
  события в источнике данных, 509
  список как приемник и источник (пример), 512
буферы, в интерпретаторе Node, 324

В

ввод/вывод, асинхронный
  в интерпретаторе Node, 321
  модуль утилит клиента HTTP в Node, 329
ввод пользователя, фильтрация, 516
веб-броузеры
  box-sizing, свойство, 457
  children, свойство объекта Element, поддержка, 402
  CORS (Cross-Origin Resource Sharing), заголовки, 545
  JavaScript в веб-броузерах, 333
  Navigator, объект, 374
  блочная модель CSS, 457
  веб-сайты с информацией о совместимости, 353
  доступность веб-приложений, 360
  значения свойства readyState объекта XMLHttpRequest, 532
  информация о броузере и об экране, 374
  использование вызываемых объектов, 214
  как операционные системы, 336
  область видимости localStorage, 630
  область видимости sessionStorage, 630
  поддержка
    CSS, 447
    SVG, 666
    баз данных, 628, 747
    объекта EventSource, 552
    форматов обозначения цветов в CSS, 458
    элемента <canvas>, 672
  проблемы совместимости, 352
  различия в реализации операций композиции, 700
  реализация модуля CSS Transitions, 467
  реализация событий, возникающих в ходе
    выгрузки файлов, 542
    выполнения HTTP-запроса, 541
  редактирование содержимого документа, 441
  текущие версии, 355
веб-документы и JavaScript, 336
веб-жучки (web bugs), 525
веб-приложения, 32
  автономные, 650
  и JavaScript, 336
  калькулятор платежей по ссуде, 33
веб-сокеты, 755
  сервер на основе веб-сокетов и интерпретатора Node, 758

- создание и регистрация обработчиков событий, 755
- создание клиента чата, 757

- веб-страницы, 32
- веб-шрифты, 450
- вещественные литералы, 52
- вещественные числа, двоичное представление вещественных чисел и ошибки округления, 55
- взаимодействие документов с разным происхождением, 716
- взаимодействие между окнами в JavaScript, 387
- видимая область
  - определение, 421
  - определение размеров, 423
- видимость
  - visibility, свойство, 457, 458
  - частичная видимость, свойства overflow и clip, 459
- визуальные эффекты
  - в библиотеке jQuery, 978
  - методы объекта jQuery, 586
  - реализация собственных анимационных эффектов, 589
  - реализация с помощью библиотеки jQuery
    - простые эффекты, 588
  - эффект смены изображений, 656
- вкладки в окнах броузеров, 382
- вложенные функции, 188, 335
- возбуждение событий, 582
  - предотвращение возбуждения событий архитектуры Ajax, 610
- возврат каретки, 42
- возвращаемые значения
  - обработчиков событий, 495
  - обработчиков событий в библиотеке jQuery, 577
  - установка свойства returnValue в значение false, 497
- восьмеричные значения, 52
- временная файловая система, 742
- всплывающие окна, блокирование в веб-броузерах, 384
- всплытие, 348, 477
  - динамических событий, 585
  - невсплывающие версии событий мыши в IE, 484
  - обработка событий в библиотеке jQuery, 575
  - событий клавиатуры в документе и окне, 484
  - событий мыши, 485
- вспомогательные технологии, 360
  - конфликт с анимационными эффектами, 588
- вспомогательные функции в библиотеке jQuery, 982
- встраивание JavaScript в разметку HTML, 337
  - <script>, элемент, 338
  - обработчики событий, 341
  - сценарии во внешних файлах, 339
- встроенные стили, управление, 463
- встряхивание элемента из стороны в сторону (пример анимационного эффекта), 465
- выбор элементов документа, 393
  - document.all[], коллекция, 400
  - по значению атрибута id, 393
  - по значению атрибута name, 394
  - по классу CSS, 397
  - по типу, 395
- вывод, определение функции jQuery.fn.println(), 622
- выделенные фоновые потоки выполнения, 724
- выделенный текст, в документах, получение, 440
- вызов конструктора базового класса, 252
- вызов конструктора и методов базового класса, 254
- вызов конструкторов, 192
- вызов методов, 190
- вызов функций, 189
  - jQuery(), функция, 558
  - вызов конструкторов, 192
  - как инструкции-выражения, 110
  - косвенный, 193
  - обработчиков событий, 492
- вызываемые объекты, 214
- выполнение JavaScript-программ, 344
  - модель потоков выполнения, 349
  - последовательность выполнения, 350
  - синхронные, асинхронные и отложенные сценарии, 345
  - управляемое событиями, 347
- выполнение итераций
  - с помощью циклов for/in, 122
- выполнение, управляемое событиями, 347
- выражение инициализации (циклы for), 121
- выражение инкремента (циклы for), 121
- выражение проверки (циклы for), 121
- выражения, 79
  - delete, оператор, 107
  - typeof, оператор, 105
  - void, оператор, 108
  - арифметические, 88
    - + (сложение и конкатенации строк), оператор, 89
  - вызова, 83, 189, 190
    - вызов конструкторов, 192
    - косвенный вызов, 193
  - вычисление, 102
    - eval(), функция, 103
  - генераторы, 308
  - доступа к свойствам, 191
    - вызов методов, 191
  - инициализаторы объектов и массивов, 80
  - использование eval() в глобальном контексте, 103
  - использование eval() в строгом режиме, 104
  - левосторонние, 86
  - логические, 98
    - логическое ИЛИ (||), оператор, 99
    - логическое И (&&), оператор, 98
    - логическое НЕ (!), оператор, 100
  - обращения к свойствам, 82
  - оператор «запятая» (,), 108
  - операторы, типы данных операндов, 86
  - определений функций, 81, 185
    - в сравнении с инструкциями объявления функций, 114
  - отношений, 93
    - оператор in, 97
    - оператор instanceof, 97
    - операторы равенства и неравенства, 93
    - операторы сравнения, 95
  - первичные, 79
  - перед ключевым словом for в генераторах массивов, 308
  - поразрядные операторы, 91

  порядок вычисления, 88
  присваивания, 100
    побочные эффекты, 110
    с операцией, 101
  создания объектов, 84
  унарные арифметические операторы, 90
  условный оператор (?:), 105
вычисление выражений, 102
  eval(), функция, 103
  использование eval() в глобальном контексте, 103
  использование eval() в строгом режиме, 104
вычисленные стили, 468, 469, 566
  получение, 468
вычисляемые значения свойств размеров в CSS, 457

Г

генераторы, 303
  конвейеры, 305
  перезапуск с помощью методов send() и throw(), 306
генераторы массивов, 307
  синтаксис, 307
геометрия документа и элементов и прокрутка, 421
  координаты документа и видимой области, 421
  обработка событий колесика мыши (пример), 506
  определение геометрии элемента, 423
  определение элемента в указанной точке, 424
  прокрутка, 425
  размеры, позиции и переполнение элементов, 426
  чтение и запись параметров геометрии элементов, 568
геопозиционирование, 707
гиперссылки
  Link, объект, 984
  onclick, обработчик событий, 434
глобальные переменные, 45, 51
  как свойства глобального объекта, 77
  ограничение на использование в безопасных подмножествах, 292
глобальные свойства, 815
глобальные функции, 45, 815
  в Rhino, 315
глобальный объект, 63, 815
  свойства, ссылающиеся на предопределенные объекты, 815
глобальный поиск по шаблону, 285, 286, 289
градиенты
  CanvasGradient, объект, 890
граница неслова () метасимвол, 284
граница слова () символ в регулярных выражениях, 284
графика, 672, 673
  SVG (Scalable Vector Graphics – масштабируемая векторная графика), 665
  работа с готовыми изображениями, 656
  создание с помощью элемента <canvas>, 672
графические атрибуты объекта контекста, 678
графические интерфейсы пользователя
  создание графического интерфейса средствами языка Java на языке JavaScript в Rhino (пример), 319
графические параметры (<canvas>), 894
  сохранение, 894
грубое определение типа, 235
группировка в регулярных выражениях, 281, 282

Д

дата и время, 56
  Date, класс, 50
    toString(), метод, 72
    valueOf(), метод, 72
  Date, объект, 781
    getDate(), метод, 785
    getDay(), метод, 785
    getFullYear(), метод, 785
    getHours(), метод, 786
    getMilliseconds(), метод, 786
    getMinutes(), метод, 786
    getMonth(), метод, 786
    getSeconds(), метод, 786
    getTime(), метод, 787
    getTimezoneOffset(), метод, 787
    getUTCDate(), метод, 787
    getUTCDay(), метод, 788
    getUTCFullYear(), метод, 788
    getUTCHours(), метод, 788
    getUTCMilliseconds(), метод, 788
    getUTCMinutes(), метод, 788
    getUTCMonth(), метод, 789
    getUTCSeconds(), метод, 789
    getYear(), метод, 789
    now(), метод, 789
    parse(), метод, 790
    prototype, свойство, 140
    setDate(), метод, 790
    setFullYear(), метод, 791
    setHours(), метод, 791
    setMilliseconds(), метод, 792
    setMinutes(), метод, 792
    setMonth(), метод, 792
    setSeconds(), метод, 793
    setTime(), метод, 793
    setUTCDate(), метод, 793
    setUTCFullYear(), метод, 794
    setUTCHours(), метод, 794
    setUTCMilliseconds(), метод, 795
    setUTCMinutes(), метод, 795
    setUTCMonth(), метод, 795
    setUTCSeconds(), метод, 796
    setYear(), метод, 796
    toDateString(), метод, 796
    toGMTString(), метод, 797
    toISOString(), метод, 797
    toJSON(), метод, 797
    toLocaleDateString(), метод, 798
    toLocaleString(), метод, 798
    toLocaleTimeString(), метод, 799
    toString(), метод, 799
    toTimeString(), метод, 799
    toUTCString(), метод, 800
    UTC(), метод, 800
    valueOf(), метод, 801
    методы, 782
    преобразование в строки и в числа, 73
    сериализация в формат ISO, 162
    статические методы, 784
двоичное представление вещественных чисел, 55

двоичные данные
  в HTTP-ответах, 535
  двоичные объекты и их прикладные интерфейсы, 732
двоичные объекты, 732, 886
  URL-адреса, отображение изображений, 736
  загрузка, 735
  использование, 734
  как файлы, 734
  конструирование, 736
  чтение, 739
двухместные операторы, 86
действия по умолчанию, для событий, 478
деление на ноль, 54
дескрипторы свойств, 154
  __proto__, свойство, 159
  вспомогательные свойства в ECMAScript 5, 268
  вспомогательные функции, 264
  получение для именованных свойств объектов, 155
диалоги, 377
  отображение разметки HTML в диалоге с помощью метода showModalDialog(), 378
динамические события, 584
директивы, 133
длина строки, 56
  определение, 59
документы, 390
  write(), метод, 438
  writeln(), метод, 439
  атрибуты элементов, 405
  вложенные HTML-документы, 382
  выбор элементов, 393
  генерирование содержимого в процессе загрузки, 345
  геометрия документа и элементов и прокрутка, 421
  загрузка нового документа, 372
  изменение структуры с помощью методов объекта jQuery, 571
  как деревья элементов, 402
  обзор модели DOM, 390
  получение выделенного текста, 440
  происхождение, 631
  редактируемое содержимое, 441
  свойства объекта Document, 437
  связывание с таблицами стилей, 446
  события загрузки, 498
  содержимое элемента, 409
  создание и вставка узлов, 413
    использование объектов DocumentFragment, 416
    удаление и замена узлов, 415
  создание оглавления (пример), 418
  структура и навигация, 401
    документы как деревья узлов, 401
  элементы документа как свойства окна, 380
домены
  domain, атрибут cookies, 636
доступ к свойствам и наследование, 145
доступность, 360
дочерние окна и история посещений, 373
древовидная структура
  геометрия документа и элементов и прокрутка, 421
  документы как деревья элементов, 402
  представление HTML-документов, 391

Ж
жадное повторение, 280

З
забоя символ в регулярных выражениях, 279
заголовки, HTTP-запросы и ответы, 528
  Content-Length, заголовок, 542
  CORS (Cross-Origin Resource Sharing), заголовки, 545
  выгрузка файлов посредством POST-запросов, 540
  заголовки в ответе, 531
  проверка заголовка Content-Type в ответе, 535
  установка заголовка Content-Type POST-запроса, 536
  установка заголовков в запросе, 530
загрузка содержимого документа, 344
заимствование методов, 247
закрытие окон, 385
заливка
  области отсечения, 693
  цвет, градиенты и шаблоны, 687
  цветом, градиентом и шаблонами в объекте Canvas, 891
замыкания, 203
  в методах чтения и записи свойств, 206
  доступ к аргументам внешней функции, 208
  использование в функции uniqueInteger() (пример), 205
  использование методов доступа к свойствам, 249
  лексические, краткая форма записи функций, 309
  правила лексической области видимости для вложенных функций, 204
запросы и ответы
  декодирование ответа, 534
  синхронные ответы, 533
запросы и ответы (HTTP), 529
  выполнение запроса, 529
  компоненты ответа, 531
  оформление тела HTTP-запроса, 535
  порядок следования частей запроса, 531
  прерывание запросов и предельное время ожидания, 544
запросы с данными в формате HTML-форм, 536
  выгрузка файлов, 539
  выполнение GET-запросов, 537
  выполнение POST-запросов, 537
  запросы с MIME-типом multipart/form-data, 540
  кодирование объектов, 536
запросы с данными в формате JSON, 538
запросы с данными в формате XML, 538
зарезервированные слова, 44
  имена HTML-атрибутов, 406
  использование в качестве имен свойств, 139
    получение и изменение значений свойств, 142
  первичные выражения, 80
зафиксированные объекты, 847, 849
знаки препинания в регулярных выражениях, 278
значение бесконечности, 53

И

- идентификаторы
  - в выражениях обращения к свойствам, 82
  - в помеченных инструкциях, 124
  - и символы управления форматом, 42
  - определение, 44
  - фрагментов в URL-адресах, 711
- изменяемые типы данных, 51
- изображения
  - Image, объект, 962
  - вставка в холст, 696
  - извлечение содержимого холста в виде изображения, 698
  - ненавязчивая реализация смены изображений, 657
  - обработки в фоновых потоках выполнения (пример), 725
  - отображение с использованием URL-адресов двоичных объектов, 736
  - рисование в элементе <canvas>, 892
- имена
  - CSS-свойств в JavaScript, 464
  - окон, 383
  - свойств объектов и индексы в массивах, 166
  - тегов, выбор элементов документа, 395
  - функций, 186
- индексы в массивах, 164
  - и имена свойств объектов, 166
- инициализаторы, выражения, 80
- инструкции, 109
  - debugger, 132
  - with, 131
  - выражения, 109, 110
  - завершение, необязательные точки с запятой, 46
  - краткая сводка, 135
  - объявления, 109, 112
    - var, инструкция, 112
  - объявления функций, 186
  - определение, 27
  - определения функций, 113
  - перехода, 109, 124, 125
    - break, инструкция, 125, 126
    - continue, инструкция, 126, 128
    - return, инструкция, 127, 129
    - throw, инструкция, 128, 129
    - try/catch/finally, инструкция, 129, 130
    - метки, 124, 125
  - помеченные, 124
    - continue, инструкция, 126
  - прочие, 131
  - составные и пустые, 110
  - условные, 114
    - else if, инструкция, 116
    - if, инструкция, 114
    - switch, инструкция, 117
  - циклов
    - do/while, инструкция, 120
    - for/in, инструкция, 122
    - for, инструкция, 121
    - while, инструкция, 119
  - циклы, 119
- интерпретаторы
  - версии JavaScript, 297
  - поддержка расширений JavaScript, 290
  - поддержка расширения E4X (ECMAScript for XML), 310
- исключения
  - Java, обработка как JavaScript-исключений в Rhino, 318
  - возбуждаемые объектом Worker, 722
  - возбуждение, 128
  - обработка с множественными блоками catch, 309
  - обработка с помощью инструкции try/catch/finally, 129
- история посещений, 373
  - History, объект, 958
  - механизм управления историей посещений в HTML5, 487, 711
- итераторы, 301
- итерации
  - по полям, методам и свойствам Java-классов и объектов, 317
  - по свойствам с помощью функции jQuery.each(), 610
  - расширения языка
    - выражения, 308
    - генераторы, 303
    - генераторы массивов, 307
- итерируемые объекты, 302

К

- калькулятор платежей по ссуде (пример), 33
- карринг, 211
- карты, отображение с использованием механизма геопозиционирования, 708
- квантификаторы регулярных выражений, 279
- клавиши-модификаторы, для событий мыши, 483, 500
- классификация броузеров, 356
- классы, 50, 221
  - className, свойство, 406
  - абстрактные, 252
  - в ECMAScript 5, 262
    - дескрипторы свойств, 268
    - определение неизменяемых классов, 263
    - определение неперечислимых свойств, 262
    - подклассы, 267
    - предотвращение расширения классов, 266
    - сокрытие данных объекта, 265
  - в стиле Java в языке JavaScript, 227
    - Set, класс (пример), 238
    - возможности, не поддерживаемые в JavaScript, 230
    - определение класса Complex, 228
    - типы-перечисления (пример), 239
    - функции, определяющие простые классы, 228
  - встроенные, автоматически определяются во всех окнах, 388
  - и конструкторы, 223
    - и идентификация классов, 225
    - свойство constructor, 226
  - и прототипы, 222
  - объектно-ориентированное программирование в JavaScript
    - наследование в сравнении с композицией, 256
    - отделение интерфейса от реализации, 258
  - определение CSS-свойств, 448

  определение в нескольких окнах, 388
  определение классов объектов, 232
    грубое определение типа, 235
    использование имени конструктора как идентификатора класса, 234
    с помощью оператора instanceof, 232
  подклассы, 252
    иерархии классов и абстрактные классы, 258
    композиция в сравнении с наследованием, 256
    определение подкласса, 252
  приемы объектно-ориентированного программирования в JavaScript
    заимствование методов, 247
    перегрузка конструкторов и фабричные методы, 250
    частные члены, 249
  расширение добавлением методов в прототип, 231
  управление CSS-классами, 470
  чтение и запись CSS-классов, 566
клиентский JavaScript, 29, 333, 334
  безопасность, 361
    атаки типа отказ в обслуживании, 367
    межсайтовый скриптинг, 365
    неподдерживаемые возможности, 362
    ограничение возможностей, 362
    политика общего происхождения, 363
  конструирование веб-приложений, 367
  модель потоков выполнения, 349
  проблемы совместимости, 352
  справочник, 881
ключевые слова
  чувствительность к регистру, 41
кнопки
  <button>, элемент, 428
    onclick, обработчик событий, 343
    регистрация обработчиков событий click, 491
  Button, объект, 888
  кнопки в формах, 433
  переключатели и флажки, 434
кодовые пункты Юникода, 57, 515
команды
  ConsoleCommandLine, объект, 907
  редактирования текста, 443
комментарии, 25
  Comment, узел, 905
  createComment(), метод объекта Document, 413
  в JavaScript-коде в URL-адресах, 342
  в таблицах стилей CSS, 445
  создание узлов Comment, 918
  стили оформления, 43
  условные, 359
композиция
  в сравнении с наследованием, 256
  в элементе <canvas>, 893
  пикселов в холсте, 698
    локальная и глобальная, 700
компоненты редакторов в фреймворках, 443
конвейеры генераторов, 305
конечные числа, 817
конкатенация строк, 59
константы, область видимости блока и использование ключевого слова let, 295
конструкторы, 140, 185, 815
  constructor, свойство объектов, 840
  prototype, свойство, 158
  вызов, 192
  вызов конструктора базового класса, 254
  и прототипы объектов, 140
  классы, 223
    и идентификация классов, 225
    свойство constructor, 226
  объектов, 228
  определение, 50
  перегрузка, 250
  типизированных массивов, 1018
контекст вызова, 185
контекст выполнения, 382
  обработчиков событий, 493
контекстные меню, 500
контекст просмотра, 382
контуры
  атрибуты рисования линий, 691
  определение, 675
  определение попадания точки в контур, 900
  холст, определение внутренней области контура, 677
  элемент <canvas>
    beginPath(), метод, 897
    closePath(), метод, 898
    создание и отображение, 890
координаты
  Geocoordinates, объект, 955
  документа и видимой области, 421
  преобразование координат в видимой области в координаты документа, 426
  преобразование координат документа в координаты видимой области, 427
  указателя мыши в системе координат окна, 483
копии, структурированные, 712
косвенный вызов, 193
кривые Безье
  quadraticCurveTo(), метод, 902
  рисование и заливка в холсте, 685
круговая диаграмма в формате SVG, построенная JavaScript-сценарием, 667
курсор, открытие в IndexedDB, 748
кэширование, 643
  мемоизация функций, 220
кэш приложений, 643
  значения свойства status, 649
  обновление кэша, 645
  создание файла объявления, 643

Л
левосторонние выражения, 86, 122
лексическая область видимости, 203
  и функции, совместно используемые в окнах или фреймах, 388
  правила для вложенных функций, 204
лексическое замыкание, 309
линейные градиенты, 690, 898
линии, атрибуты рисования в холсте, 691
литералы, 80
  вещественных чисел, 52
  литералы XML в программном коде JavaScript, 311
  объектов, 139
  определение, 43
  строковые, 56
  числовые, 52

- логические выражения, 98
  - логическое ИЛИ (||), оператор, 99
  - логическое И (&&), оператор, 98
  - логическое НЕ (!), оператор, 100
- логические значения, 49, 61
  - преобразование объектов в логические значения, 71
- локальные файлы и объект XMLHttpRequest, 529

М

- массивы, 50, 164
  - Array, класс, 50
    - toString(), метод, 71
  - Array, объект, 763
    - concat(), метод, 173, 765
    - every(), метод, 178, 765
    - filter(), метод, 177, 766
    - forEach(), метод, 176, 767
    - indexOf(), метод, 180, 768
    - isArray(), метод, 181
    - join(), метод, 172, 769
    - lastIndexOf(), метод, 180, 769
    - length, свойство, 764, 770
    - map(), метод, 177, 770
    - pop(), метод, 175, 771
    - push(), метод, 175, 771
    - reduce(), метод, 178, 772
    - reduceRight(), метод, 178, 773
    - reverse(), метод, 172, 774
    - shift(), метод, 175, 774
    - slice(), метод, 174, 775
    - some(), метод, 178, 775
    - sort(), метод, 173, 776
    - splice(), метод, 174, 777
    - toLocaleString(), метод, 176, 778
    - toString(), метод, 176, 778
    - unshift(), метод, 175, 779
    - методы, 764
  - Java, получение и изменение элементов в программах на языке JavaScript Rhino, 317
  - sort(), метод, 247
  - выражения обращения к свойствам, 82
  - длина, 168
  - добавление и удаление элементов, 169
  - значений, присваивание с разложением, 298
  - инициализаторы, 80
  - итерации
    - с помощью метода jQuery.each(), 610
    - с помощью циклов for/each, 300
  - методы, 172
    - ECMAScript 5, 176
  - многомерные, 171
  - обработка с помощью функций, 215
  - обход элементов массива, 170
  - объекты как ассоциативные массивы, 143
  - отличие от других объектов, 181
  - поиск по строкам с помощью регулярных выражений, 286
  - преобразование объекта jQuery в настоящий массив, 562
  - присваивание функций элементам, 199
  - разреженные, 167
  - создание, 165
  - сравнение, 66
  - строки как массивы, 184
  - типизированные, 1017
  - типизированные массивы и буферы, 728
    - разновидности типизированных массивов, 728
  - чтение и запись элементов массивов, 166
- междоменные HTTP-запросы, 545
  - получение информации о ссылках с помощью HEAD-запросов при поддержке заголовка CORS, 546
- межсайтовый скриптинг (XSS), 365
- мемоизация функций, 220
- метасимволы в регулярных выражениях, 277
- методы, 64, 65, 185
  - Java, вызов из программ на языке JavaScript в Rhino, 316
  - jQuery (термин), 561
  - абстрактные, 252
  - вставки и удаления в библиотеке jQuery, 974
  - выбора в библиотеке jQuery, 618, 969
    - возврат к предыдущему выбору, 621
    - использование результатов выбора в качестве контекста, 619
  - вызов, 190
  - выражения вызова, 83
  - добавление в объекты-прототипы, 231
  - заимствование, 247
  - класса Object, 162
    - toJSON(), 163
    - toLocaleString(), 163
    - toString(), 162
    - valueOf(), 163
  - массивов
    - ECMAScript 3, 172
    - ECMAScript 5, 176
  - массивов, введенные стандартом ES5, 356
  - ограничение на использование в безопасных подмножествах, 293
  - преобразования типов, 242
  - сравнения, 244
  - чтения и записи
    - объекта jQuery, 565
      - CSS-атрибутов, 566
      - CSS-классов, 566
      - HTML-атрибутов, 565
      - данных элемента, 570
      - значений элементов HTML-форм, 567
      - параметров геометрии, 568
      - содержимого элементов, 568
    - свойств, 152
- механизм буксировки (drag-and-drop)
  - API, определяемый стандартом HTML5, 486
- многомерные массивы, 171
- многострочный режим поиска по шаблону, 285
- многоугольники, рисование методами объекта Canvas, 675
- множества
  - Set, класс (пример), 238, 239
- мобильные устройства, события, 479
- модальные диалоги, 378
- модель выполнения фоновых потоков, 723
- модули, 270, 273
  - область видимости функции как частное пространство имен, 273
  - объекты как пространства имен, 271
- мышь
  - двунаправленное колесико мыши и событие wheel, 485

  - координаты указателя мыши, 422

Н

- набор символов, 41
- наиболее важные CSS-свойства, 450
- наследование, 144
  - в сравнении с композицией, 256
  - и перечисление свойств, 149
  - классы и прототипы, 222
  - прототип конструктора как прототип нового объекта, 223
  - свойств объектов, 144
  - свойств с методами чтения и записи, 152
  - создание новых объектов, наследующих прототипы, 141
- нежадное повторение, 280
- неизменяемые классы, определение, 263
- неизменяемые объекты, 847, 849
- неизменяемые типы данных, 51
- неопределенные значения, 879
  - переменные, объявленные без инициализаторов, 112
- неполная иерархия классов узлов документов, 392
- нестандартные CSS-свойства, 447
- нормализация способов кодирования Юникода, 43

О

- области видимости блока, 76
  - отсутствие, решение проблемы с помощью ключевого слова let, 295
- области видимости переменных, 51, 75
  - cookies, 635
  - вложенные функции, 188
  - замыкания, 203
  - и функции, 185
  - область видимости хранилища, 630
  - фонового потока, 722
  - функции как пространства имен, 76, 201
  - функции обработчиков событий, 494
  - цепочки областей видимости, 78
- области видимости функций, 51
  - и подъем, 76
  - как частные пространства имен модулей, 273
- обмен сообщениями
  - message, событие, используемое для обмена асинхронными сообщениями, 487
  - отправка данных объекту Worker с помощью метода postMessage(), 721
- обработка исключений, 129
- обработка событий колесика мыши, 504
- обработчики событий
  - ApplicationCache, объект, 884
  - Element, объект, 931
  - EventSource, объект, 944
  - FileReader, объект, 948
  - Form, объект
    - методы, 951
  - FormControl, объект, 953
  - MediaElement, объект, 990
  - Window, объект, 1028
  - Worker, объект, 1030
  - XMLHttpRequest, объект, 1038
    - события readystatechange, 532
  - в HTML, 341
  - в библиотеке jQuery, 577
  - возникающих в ходе выполнения HTTP-запроса, 541
  - возникающих в ходе выполнения выгрузки файлов, 542
  - вызов, 492
    - аргумент обработчика события, 493
    - возвращаемое значение обработчика, 495
    - контекст обработчика события, 493
    - отмена событий, 497
    - порядок вызова, 495
  - дополнительные способы регистрации обработчиков событий в библиотеке jQuery, 579
  - область видимости, 494
  - определение, 31
  - определение в автономных веб-приложениях, 651
  - определение в объекте FileReader, 740
  - определение обработчика onclick (пример), 31
  - распространение событий, 496
  - свойства, определяемые объектами HTMLElement, 405
  - текстовых полей ввода, 435
  - удаление в библиотеке jQuery, 580
  - форм и их элементов, 432
  - функции, 348
- обратный порядок следования байтов (little-endian), 731, 913
- обход элементов массива, 170
- объектно-ориентированное программирование
  - в JavaScript
    - наследование в сравнении с композицией, 256
    - отделение интерфейса от реализации, 258
  - строго типизированные языки, 221
- объектные базы данных, 748
- объектные типы, 49
- объекты, 137
  - instanceof, оператор, 97
  - Java, изменение значений полей в Rhino, 317
  - jQuery, 561
  - XML, 310
  - атрибуты, 158, 159
    - class, атрибут, 159
    - extensible, атрибут, 160
    - prototype, атрибут, 158
  - атрибуты свойств, 154
  - базового языка, 138
  - вызываемые, 214
  - выражения обращения к свойствам, 82
  - выражения создания объектов, 84
  - глобальный объект, 63
  - инициализаторы, 80
  - итерации по свойствам
    - порядок перечисления в циклах for/in, 123
    - с помощью циклов for/in, 122
  - итерируемые, 302
  - как пространства имен, 271
  - конструкторов, 227
  - контекста рисования, 673
  - массивы объектов, 164
  - методы, 185
  - методы чтения и записи свойств, 152
  - обертки, 64
  - определение классов объектов, 232
    - грубое определение типа, 235
    - использование имени конструктора как идентификатора класса, 234
    - с помощью оператора instanceof, 232

  - перечисление свойств, 149
  - подобные массивам, 182
    - Arguments, объект, 194
    - HTMLCollection, объект, 396
    - jQuery, 562
    - типизированные массивы, 728
  - получение и изменение значений свойств, 142
    - и наследование, 144
    - объекты как ассоциативные массивы, 143
    - ошибки доступа, 145
  - представление свойств в формате HTML-форм, 536
  - преобразование в строки, 712
  - проверка существования свойств, 148
  - прототипов, 228
  - свойства, 138
  - свойства с методами доступа, 152
  - сериализация, 161
  - создание, 139, 141
    - прототипы, 141
    - с помощью литералов, 139
    - с помощью оператора new, 140
    - с помощью функции Object.create(), 141
  - среды выполнения, 138
  - ссылки на изменяемые объекты, 65
  - структурированные копии, 712
  - удаление свойств, 147
  - экземпляров, 228
- объявление переменных, 74
- объявление функций, 187
- оглавление, пример создания оглавления документа, 418
- окна, 369
  - адрес документа и навигация по нему, 371
    - загрузка нового документа, 372
  - взаимодействие JavaScript-кода с окнами, 363
  - взаимодействие между окнами в JavaScript, 387
  - закрытие окон, 385
  - и история посещений, 373
  - информация о броузере и об экране, 374
  - обработка ошибок, 379
  - определение позиций полос прокрутки, 422
  - определение размеров видимой области, 423
  - открытие и закрытие, 383
  - политика общего происхождения, document.domain, свойство, 364
  - работа с несколькими окнами и фреймами, 382
  - таймеры, 370
  - элементы документа как свойства окна, 380
- операторы, 84, 86
  - delete, 107
  - in, 97
  - instanceof, 97
  - typeof, 105
  - void, 108
  - + (сложение и конкатенации строк), оператор, 89
  - арифметические, 88
  - ассоциативность, 88
  - двухместные, 86
  - «запятая» (,), 108
  - количество операндов, 86
  - левосторонние выражения, 86
  - логическое И (&&), 98
  - логическое ИЛИ (||), 99
  - логическое НЕ (!), 100
  - обзор операторов, 84
  - операнды, 86
  - отношений, 93
  - перечень операторов JavaScript, 84
  - побочные эффекты, 87, 110
  - поразрядные, 91
  - порядок вычисления, 88
  - приоритет, 87
  - равенства и неравенства, 93
  - сдвига, 92
  - сравнения, 95, 246
  - тернарные, 86
  - типы данных операндов, 86
  - унарные, 86
  - унарные арифметические операторы, 90
  - условный оператор (?:), 105
- операции чтения, 948
- операционные системы, основа реализации механизмов буксировки (drag-and-drop), 486
- определение класса объекта, 159
- определение собственных свойств функций, 200
- определение типа броузера, 374
  - с помощью navigator.userAgent, 376
- определение элемента в указанной точке, 424
- отключение анимационных эффектов в библиотеке jQuery, 588
- открывающие и закрывающие теги, 410
- открытие и закрытие окон, 383
  - закрытие окон, 385
- отладка фоновых потоков выполнения, 727
- отложенное выполнение сценариев, 346
- отмена событий, 497
  - textinput и keypress, 516
- относительное позиционирование элементов, 452
- относительные URL-адреса, 373
- относительные значения, для числовых свойств при воспроизведении анимационных эффектов, 590
- отрицания символ в классах символов, 278
- отрицательная бесконечность, 53
- отрицательный ноль, 54
  - сравнение с положительным нулем, 54
- отступы
  - в блочной модели CSS, 455
  - определение средствами CSS, 454
- отсчет индексов с 0 (в массивах и строках), 164
- оформление тела HTTP-запроса, 535
- ошибки
  - Error, класс, 50, 130
  - Error, объект, 804
    - javaException, свойство, 318
    - message, свойство, 805
    - name, свойство, 805
    - toString(), метод, 806
  - в броузерах, 353
  - доступа к свойствам, 145
  - обработка в объектах Window, 379
  - обработка ошибок подобно событиям, 481
  - округления вещественных чисел в двоичном представлении, 55

П

- параметры (функций), 193, 194
  - необязательные, 193
- первичные выражения, 79

- перегрузка конструкторов, 250
- перекрытие полупрозрачных окон (пример), 460
- переменные
  - в генераторах массивов, 307
  - глобальные, 45
  - и типы данных, 51
  - как свойства, 77
  - область видимости, 75
    - функции как пространства имен, 76
    - цепочки областей видимости, 78
  - объявление, 74
    - повторные и опущенные объявления, 74
    - с помощью ключевого слова let, 295
  - присваивание с разложением, 298
  - присваивание функций, 199
  - ссылка на переменную как первичное выражение, 80
- переполнение, 53
  - элементов, прокручиваемые элементы документа, 427
- перехват событий, 478, 491, 496
  - использование метода setCapture() в IE для перехвата событий мыши, 501
- перехватывающие обработчики событий, 491, 496
  - события мыши в IE, 501
- перечисление свойств, 149
  - порядок в циклах for/in, 123
- перечислимые свойства, 123
  - итерации с помощью циклов for/each, 300
  - получение имен собственных перечислимых свойств, 849
- печать, события beforeprint и afterprint, 488
- пикселы
  - композиция в холсте, 698
  - композиция в элементе <canvas>, 893
  - манипулирование в объекте Canvas, 894
- побочные эффекты выражений, 110
- повторение в регулярных выражениях, 279
  - жадное, 280
  - нежадное, 280
- поддержка метода getElementsByClassName(), 398
- поддержка селекторов CSS, 398
- подклассы, 252
  - вызов конструктора и методов базового класса, 254
  - иерархии классов и абстрактные классы, 258
  - композиция в сравнении с наследованием, 256
  - определение, 252
- подмножества JavaScript, 291
  - ADsafe, 293
  - Caja, 294
  - dojox.secure, 294
  - FBJS, 294
  - Microsoft Web Sandbox, 295
  - The Good Parts, 291
- подстроки, разбиение строк, 287
- подшаблоны в регулярных выражениях, 281
- подъем, 76
- подынструкции, 111
- позиции полос прокрутки окна, 422
- позиционирование элементов
  - блочная модель CSS, 455
  - геометрия документа и элементов и прокрутка, 421
  - с помощью CSS, 451
  - текст с тенью (пример), 454
  - чтение и запись параметров геометрии элементов, 568
- по значению
  - сравнение значений простых типов, 66
- поиск с заменой с использованием регулярных выражений, 285
- политика общего происхождения, 363
  - и вызываемые ею проблемы, 364
  - и файловые системы, 742
  - объект Canvas, метод toDataUrl(), 698
  - предотвращение обмена cookies между сайтами, 636
  - происхождение документа, 631
- положительная бесконечность, 53
- полупрозрачные окна, перекрытие (пример), 460
- получение, вставка и удаление правил из таблиц стилей, 473
- пользовательские объекты, 138
- пользовательский интерфейс
  - jQuery UI, библиотека, 625
  - события, 480
- поля
  - в блочной модели CSS, 455
  - определение средствами CSS, 454
- поля и методы класса, 227
- поля и методы экземпляров, 227
- помеченные инструкции, 124
  - continue, инструкция, 126
- поразрядные операторы, 91
- порядок вызова обработчиков событий, 495
- порядок вычисления операторов, 88
- по ссылке, сравнение объектов, 66
- постоянная файловая система, 742
- постфиксный оператор декремента, 91
- постфиксный оператор инкремента, 91
- потеря значащих разрядов, 54
- потоки ввода/вывода, в Node, 324
- потоки выполнения в клиентском JavaScript, 349
- потоковые мультимедийные данные, свойство initialTime, 661
- правила стилей, 445, 908
  - получение, вставка и удаление, 473
- правило ненулевого числа оборотов, 677
- представление элементов <style> и <link>, управление таблицами стилей, 472
- преобразование
  - setTransform(), метод, 903
  - в строки, 176
  - объектов в простые значения, 71
  - системы координат в элементе <canvas>, 892
  - системы координат холста, 680
    - математический смысл, 682
    - примеры преобразований, 683
  - типов, 67
- преобразования и равенство, 68
- префиксный оператор декремента, 91
- префиксный оператор инкремента, 91
- префиксы в именах нестандартных CSS-свойств, 447
- привлечение внимания к элементам документа, 470
- приемы объектно-ориентированного программирования в JavaScript, 238
  - Set, класс (пример), 238
  - заимствование методов, 247
  - методы сравнения, 244

  перегрузка конструкторов и фабричные методы, 250
  стандартные методы преобразований, 242
  типы-перечисления (пример), 239
  частные члены, 249
прикладной интерфейс к файловой системе, 742
прикладные события, 481
приоритет операторов, 87
присваивание
  свойствам
    правила успешного присваивания, 146
  с разложением, 298, 303
присваивания выражения, 100
  с операцией, 101
пробельные символы
  в программном коде JavaScript, 42
проверка орфографии в броузерах, 441
проверка особенностей броузеров, 357
  проверка типа, 358
проверка существования свойств, 148
программы, на языке JavaScript, 344
программы чтения с экрана, 360
продолжительность анимационных эффектов, 586
  передача методам воспроизведения анимационных эффектов, 587
прозрачность
  композиция пикселов с разным уровнем прозрачности, 699
  определение цветов в CSS, 458
происхождение документа, 363, 631
прокрутка, 425
  scroll, событие окон, 483
пространства имен
  createElementNS(), метод объекта Document, 413, 918
  getElementsByTagNameNS(), метод объекта Document, 919
  jQuery, 560
  lookupNamespaceURI(), метод объекта Node, 1001
  lookupPrefix(), метод объекта Node, 1001
  process, пространство имен в Node, 323
  SVG, 669
  для обработчиков событий в библиотеке jQuery, 580
  область видимости функции как частное пространство имен, 273
  объекты, 271
  функции, 201
простые и объектные типы данных, 49
простые типы данных, 49
  неизменяемые простые значения и ссылки на изменяемые объекты, 65
  преобразование, 67
  преобразование объектов в простые значения, 71
  преобразование простых типов JavaScript в простые типы Java, 318, 319
прототипы, 140, 228, 415, 841
  Array.prototype, 164
  jQuery.fn, 622
  и классы, 222
  и наследование, 138
  и свойство constructor, 226
  конструкторов как прототипы новых объектов, 223
  корректная инициализация, ключ к созданию подклассов, 252
  определение, 138
  предотвращение расширения классов, 266
  проверка цепочки прототипов объекта, 233
процедуры, 188
прямой порядок следования байтов (big-endian), 731, 913
прямоугольники
  clearRect(), метод, 897
  ClientRect, объект, 904
  fillRect(), метод, 899
  рисование в холсте, 687
  рисование в элементе <canvas>, 892

Р

радиальные градиенты, 690, 898
разделитель абзацев, 42
разделяемые фоновые потоки выполнения, 724
размеры холста, 679
размеры элементов
  определение с помощью CSS-свойств, 453
разреженные массивы, 164, 167
разрывы строк
  в программном коде JavaScript, 42
  интерпретация, как точек с запятой, 46
рамки
  в блочной модели CSS, 455
  определение средствами CSS, 454
  определение цвета рамки элемента, 458
распространение событий, 477, 496
  определение, 477
  остановка, 941
  отмена, 497
растворение элемента (пример анимационного эффекта), 465
расширения
  E4X (ECMAScript for XML), 310
  выражения, генераторы, 308
  генераторы, 303
  генераторы массивов, 307
  итераторы, 301
  итерации, 300
    итераторы, 301
    циклы for/each, 300
  константы и контекстные переменные, 295
  краткая форма записи функций, 309
  предотвращение расширения классов, 266
  присваивание с разложением, 298
расширяемость объектов, 847
реализация свойства outerHTML с помощью свойства innerHTML, 415
реализация типов и классов
  объекты XML и стандарт E4X, 310
регистрация обработчиков событий, 348, 489
  возникающих в ходе выполнения HTTP-запроса, 542
  для событий load и click, 31
  дополнительные способы в библиотеке jQuery, 579
  простые методы объекта jQuery регистрации обработчиков, 575
  с помощью метода addEventListener(), 491
  удаление обработчиков событий в библиотеке jQuery, 580
  установкой атрибутов, 490

  - установкой свойств, 489
- регулярные выражения, 60, 276
  - RegExp, объект, 287
  - методы класса String, 285
  - определение регулярных выражений, 276
    - альтернативы, группировка и ссылки, 281
    - задание позиции соответствия, 283
    - знаки препинания, 278
    - классы символов, 278
    - литеральные символы, 276, 277
    - специальные символы, 279
    - флаги, 284
  - определение шаблонов, 277
  - подшаблоны, 281
- редактируемое содержимое документов, 441
- режим совместимости, 457
- реляционные базы данных, 748
- решето Эратосфена, алгоритм, 729

С

- сборка мусора, 50
- свойства
  - CSS-свойства стиля, 445
  - атрибуты, 154, 155
  - выражения обращения к свойствам, 82
  - вычисленного стиля, 468
  - глобальные, 815
  - имена и значения, 139
  - использование в качестве аргументов функций, 196
  - методы чтения и записи, 152
  - наиболее важные CSS-свойства, 450
  - обработчиков событий, 341, 489
  - объекта Document, 437
  - объекта HTMLElement, отражающие HTML-атрибуты, 405
  - объектов Window, Document и Element, 335
  - ограничение на использование в безопасных подмножествах, 293
  - определение, 138
  - определение неперечислимых свойств, 262
  - определение собственных свойств объекта Function, 200
  - перечисление, 149
  - получение и изменение значений, 142
    - наследование свойств, 144
    - объекты как ассоциативные массивы, 143
    - ошибки доступа, 145
  - преобразование имен HTML-атрибутов, 406
  - преобразование имен атрибутов с данными, 407
  - присваивание функций свойствам, 199
  - проверка существования, 148
  - с данными, 152
    - атрибуты, 154
  - с методами доступа, 152
    - дескрипторы свойств, 154
    - добавление в существующие объекты, 152
    - использование, 153
    - определение с помощью литералов объектов, 152
    - устаревшие приемы работы, 157
  - сокращенная форма определения свойств в CSS, 447
  - удаление, 147
  - форм и их элементов, 431
  - функций, 209
  - холста, 894
- селекторы
  - CSS, 398, 445
    - для правил стиля, 473
    - использование в вызове функции jQuery(), 558
  - в библиотеке jQuery, 613, 968
    - группы, 617
    - комбинированные, 617
    - фильтры, 614
- серверный JavaScript, 314
  - асинхронный ввод/вывод в интерпретаторе Node, 321
  - управление Java с помощью Rhino, 315
- сериализация объектов, 161, 717
  - пример использования функции JSON.stringify(), 821
- сетевые взаимодействия
  - модуль net в интерпретаторе Node, 327
- сетевых взаимодействий API для веб-приложений, 337
- символы и кодовые пункты Юникода, 57
- символы, литеральные в регулярных выражениях, 277
- символы управления форматом (Юникод), 42
- синхронное выполнение сценария, 346
- синхронные HTTP-ответы, 533
- система координат и преобразования, 892
- система координат холста, 679
- системы организации безопасного окружения, 292
- смена изображений, 656
- снежинки Коха, рисование (пример), 683
- собственные свойства, 139
- собственные события, реализация с помощью библиотеки jQuery, 584
- события, 476
  - ErrorEvent, объект, 935
  - HashChangeEvent, объект, 957
  - HTML5, 486
  - message, 717, 721, 993
  - MessageEvent, объект, 993
  - PageTransitionEvent, объект, 1003
  - PopStateEvent, объект, 1004
  - progress, 1005
  - ProgressEvent, объект, 1005
  - storage, 1010
  - StorageEvent, объект, 1010
  - архитектуры Ajax в библиотеке jQuery, 608
  - буксировки (drag-and-drop), 508
  - в библиотеке jQuery, 976
  - ввода, аппаратно-зависимые и аппаратно-независимые, 480
  - ввода текста, 515
    - использование события propertychange для определения факта ввода текста, 518
    - фильтрация ввода (пример), 516
  - возникающие в ходе выгрузки файлов, 542
  - возникающие в ходе выполнения HTTP-запроса, 541
  - возникающие в ходе приема ответа, 532
  - вызов обработчиков событий, 492
  - действия по умолчанию, 478
  - загрузки документа, 498
  - изменения состояния, 481
  - источники событий в Node, 323
  - категории, 480

  клавиатуры, 518, 521
    keydown и keyup, 519
    Keymap, класс поддержки обработки комбинаций клавиш (пример), 520
    ввод текста, 515
  колесика мыши, 504
    обработка, 506
  модели DOM, 485
  мультимедийных элементов, 663
  мыши, 483, 485, 500
  не зависящие от типа устройства, 361
  обзор, 347
  обработка событий кэша приложений, 646, 648
  обработка с помощью библиотеки jQuery, 575
    Event, объект, 577
    возбуждение событий, 582
    динамические события, 584
    дополнительные способы регистрации обработчиков событий в библиотеке jQuery, 579
    методы регистрации обработчиков, 575
    реализация собственных событий, 584
    удаление обработчиков событий, 580
  объекта Storage, 633
  объект события, 477
  определение, 476
  определяемые стандартом Server-Sent Events для архитектуры Comet, 550
  от таймеров, обработка подобно обычным событиям, 481
  поддерживаемые объектом Document, 921
  предельное время ожидания выполнения запросов, 544
  прерывания запросов, 544
  распространение, 477, 496
  реализация Java-интерфейса приемника событий в JavaScript в Rhino, 318
  регистрация обработчиков, 489
  сенсорных экранов и мобильных устройств, 488
  старые типы событий, 479
  тип или имя события, 476
  типы, 479
  установка атрибутов обработчиков событий, 490
  цель события, 476
совместимость на стороне клиента, 352
  библиотеки обеспечения совместимости, 356
  классификация броузеров, 356
  проверка особенностей, 357
  проверка типа броузера, 358
  режим совместимости и стандартный режим, 358
  условные комментарии в Internet Explorer, 359
соглашения об именах CSS-свойств в JavaScript, 464
содержимое, простой клиентский сценарий, исследующий содержимое документа, 335
создание новых таблиц стилей, 474
сокращенная форма определения свойств в CSS, 447
сопоставление с шаблонами, 60
сортировка без учета регистра символов, массивы строк, 173
составление цепочек вызовов методов, 190
сохранение данных API для веб-приложений, 337
сохранение данных на стороне клиента, 627
  cookies, 634
    реализация хранилища, 639
  безопасность и конфиденциальность, 629
  механизм сохранения userData в IE, 641
  свойства localStorage и sessionStorage, 630
  хранилище приложений и автономные веб-приложения, 643
    объявление кэшируемого приложения, 643
списки
  аргументов переменной длины, 194
  фильтрация в расширении E4X, 312
сравнение
  и преобразование типов, 68
  объектов, 66
  простых значений, 66
сравнения методы
  реализация в классах, 244
    compareTo(), метод, 246
    equals(), метод, 244
ссылки, 66
  Link, объект, 984
  в регулярных выражениях, 281
  межсайтовый скриптинг, 365
  получение информации с помощью HEAD-запросов при поддержке заголовка CORS, 546
старые типы событий, 479
  события клавиатуры, 484
  события мыши, 483
  события объекта Window, 482
  события форм, 479
статическое позиционирование элементов, 451
стили, каскады, 446
строгий режим, 133
  зарезервированные слова, 45
строки, 56, 57
  String, объект
    объекты-обертки, 64
  доступ к свойствам с использованием формы записи [], 143
  значения в таблицах стилей или в атрибуте style, 463
  как массивы, 60, 184
  методы, использующие регулярные выражения, 285
  представление состояний приложения, 712
  преобразование массивов, 176
  преобразование между JavaScript и Java, 319
  преобразование объектов в строки, 71
  работа со строками, 59
  сопоставление с шаблонами, 60
строковые литералы, 56
  и управляющие последовательности, 58
структура документа и навигация по документу, 401
  документы как деревья узлов, 401
сценарии
  jQuery.getScript(), функция, 598
  Script, объект, 1007
  атрибут type, определяющий тип MIME, 341
  во внешних файлах, 339
  синхронные, асинхронные и отложенные, 345

Т

таблицы стилей, 472
  CSSStyleSheet, объект, 910
  включение в HTML-документы, 446
  включение и выключение, 472

  - определение, 445
  - получение, вставка и удаление правил, 473
  - создание, 474
- таймеры, 370
  - вспомогательные функции (пример), 370
  - использование для воспроизведения анимации средствами CSS, 465
- текст, 56
  - встраивание произвольных текстовых данных с помощью элемента <script>, 341
  - в элементах <script>, 411
  - получение выделенного текста, 440
  - работа со строками, 59
  - рисование в холсте, 692
  - рисование в элементе <canvas>, 892
  - содержимое элемента в виде простого текста, 410
  - сопоставление с шаблонами, 60
  - с тенью, пример позиционирования элементов, 454
  - строковые литералы, 56
  - текстовые поля ввода, 434
  - управляющие последовательности в строковых литералах, 58
  - чтение текстовых файлов с помощью объекта FileReader, 740
- тени
  - рисование в холсте, 695
  - рисование в элементе <canvas>, 893
- тернарные операторы, 86
- типизированные массивы и буферы
  - set(), метод, 729
  - subarray(), метод, 730
  - эффективность, 730
- тип (имя тега)
  - выбор элементов документа, 395
- тип события, 476, 479
- тип сценария, 340
- типы данных, 25, 49
  - Java, преобразование типов в Rhino, 318
  - аргументов функций, 197
  - вещественные числа, 51
  - изменяемые и неизменяемые, 51
  - и классы, 232
    - грубое определение типа, 235
    - заимствование методов, 247
    - методы сравнения, 244
    - определение классов объектов с помощью оператора instanceof, 232
    - стандартные методы преобразований, 242
    - частные члены, 249
  - и объявление переменных, 74
  - классы, методы, 50
  - логические значения, 61
  - операндов и результата, 86
  - определение классов объектов
    - грубое определение типа, 235
    - использование имени конструктора как идентификатора класса, 234
  - поддерживаемые реализацией Ajax в библиотеке jQuery, 601
  - преобразование, 51, 67
    - объектов в простые значения, 71
  - преобразования и равенство, 68
  - текст, 56
    - работа со строками, 59
    - сопоставление с шаблонами, 60
  - целые числа, 51
  - числа, 51
- типы-перечисления, 239
  - для представления игральных карт, 241
- транзакции, управление в IndexedDB, 749
- транспорты, 525
- третье измерение, свойство z-index, 453
- трехмерная графика в элементе <canvas>, 673

У

- узлы, 391
  - вставка узлов, 413
  - использование объектов DocumentFragment, 416
  - создание узлов, 413
  - удаление и замена узлов, 415
- унарные арифметические операторы, 90
- унарные операторы, 86
- унаследованные свойства, 139
- управление встроенными стилями
  - анимационные эффекты, 465
- управление историей посещений в HTML5, 711
- управление таблицами стилей, 472
- управляемый событиями этап выполнения программ, 345
- управляющие последовательности
  - в строковых литералах, 58
- условные инструкции, 109, 114
  - else if, инструкция, 116
  - if, инструкция, 114
  - switch, инструкция, 117
  - в генераторах массивов, 308
- условные комментарии в Internet Explorer, 359
- условный оператор (?:), 105
- установка свойств обработчиков событий, 489

Ф

- фабричные методы, 250
- фабричные функции, создание и инициализация нового объекта, 222
- файловые системы, 742
  - использование асинхронного интерфейса к файловой системе, 742
  - использование синхронного интерфейса, 746
  - работа с файлами в локальной файловой системе, 742
  - сохранение данных веб-приложений на стороне клиента, 629
- файлы
  - API к файлам и файловой системе в Node, 325
  - File, объект, 945
  - выгрузка посредством HTTP-запроса POST, 539
  - как двоичные объекты, 734
  - локальные файлы и объект XMLHttpRequest, 529
  - мониторинг хода выгрузки файлов по протоколу HTTP, 542
  - объявлений кэшируемых приложений, 643
    - сложные объявления, 644
- Фибоначчи числа, функция-генератор, 305
- фиксировано позиционированные элементы, 452
- фильтрация ввода, 516
- флаги, регулярные выражения, 284, 289
- фоновые потоки выполнения, 720
  - выполнение синхронных запросов с помощью объекта XMLHttpRequest (пример), 726

- и объект FileReader, 741
- использование синхронного интерфейса к файловой системе, 746
- область видимости, 722
- отладка, 727

форма представления чисел с фиксированной точкой, 835

формы HTML, 428
- HTTP-запросы с данными в формате HTML-форм, 536
- возвращаемое значение обработчика события, предотвращение отправки формы с неверными данными, 495
- выбор форм и элементов форм, 430
- обработчики событий форм и их элементов, 432
- свойства форм и их элементов, 431
- события, 479
- элементы, 428

фрагменты контура (объект Canvas), 675

фреймворки, клиентские, 356, 367

фреймы
- взаимодействие JavaScript-кода с фреймами, 363
- видимая область, 421
- ослабление политики общего происхождения, 364
- отношения между фреймами, 385
- работа с несколькими окнами и фреймами, 382

функции, 185
- arguments[], массив, 761
- bind(), метод, 211
- call() и apply(), методы, 210
- Function(), конструктор, 213
- jQuery(), функция ($()), 558
- toString(), метод, 213
- аргументы и параметры, 193
  - использование свойств объекта, 196
  - необязательные аргументы, 193
  - списки аргументов переменной длины, 194
  - типы данных аргументов, 197
- вложенные, 335
- возвращающие массивы, присваивание с разложением, 298
- вызов, 189
  - по мере готовности документа, 499
- вызываемые объекты, 214
- выражения вызова, 83
- выражения определений, 81
- высшего порядка, 217
- генераторы, 304
- глобальные, 45, 815
- замыкания, 203
- именование, 186
- как данные, 198
- как пространства имен, 201
- конструкторы, 815
- краткая форма записи, 309
- математические, 822
- мемоизация, 220
- обработка массивов, 215
- объявление переменных с помощью ключевого слова let, 295
- ограничение на использование в безопасных подмножествах, 293
- определение, 50, 185
  - вложенных функций, 188
- определение собственных свойств, 200
- переходов, 592
- свойства, 209
  - length, 209
  - prototype, 209, 226
- совместное использование в окнах или фреймах, 387
- утилиты поддержки Ajax в библиотеке jQuery, 597
- функция jQuery (термин), 561
- частичное применение функций, 218

функции обратного вызова, 348
- jQuery.ajax(), функция, 605
- передаваемые функциям setTimeout() и setInterval(), 349
- передача методам воспроизведения анимационных эффектов, 587

функциональное программирование, 215

Х

хранилища объектов, 748

хранилище приложений, 643

Ц

цвет
- background-color, свойство, 453
- shadowColor, свойство, 695
- определение с помощью CSS, 458
- определение цвета в холсте, 687
- определение цвета контура в элементе <canvas>, 891

целые литералы, 52

цепочки вызовов методов, 190

цепочки областей видимости, 78

циклы, 119, 120
- do/while, инструкция, 120
- for, инструкция, 121
- for/in, инструкция, 122
- while, инструкция, 119
- и инструкция continue, 127

цифры, ASCII, в регулярных выражениях, 279

Ч

частичное применение функций, 218

частные свойства, 230

частные члены, имитация в JavaScript, 249

чат
- клиент на основе веб-сокетов, 757
- простой клиент на основе объекта EventSource, 551
- сервер на основе веб-сокетов и интерпретатора Node, 758
- сервер чата, поддерживающий протокол Server-Sent Events, 554

числа, 51
- арифметические операции в JavaScript, 53
- вещественные литералы, 52
- двоичное представление вещественных чисел и ошибки округления, 55
- объекты-обертки, 64
- определение класса комплексных чисел (пример), 228
- преобразование
  - объектов в числа, 71
  - чисел в строки, 69
- целые литералы, 52

числовые типы, 49

члены класса, в строго типизированных языках, 227
чувствительность к регистру в JavaScript, 41

Ш

шаблонные URL-адреса, в объявлениях кэшируемых приложений, 645
шаблоны, использование для рисования и заливки контуров, 890
шестнадцатеричные значения, 52
шестнадцатеричные цифры, определение цветов в формате RGB, 458
шрифты
  font-size, font-weight и color, свойства стиля, 463
  веб-шрифты в CSS, 450

Э

экземпляры, 221
  constructor, свойство, 226
  свойства объекта RegExp, 287
  создание и инициализация с помощью фабричных функций, 222
  создание с помощью фабричных методов, 250
экранированные последовательности Юникода, 42
экспоненциальная форма представления, 52, 834
элементы
  display и visibility, свойства (CSS), 457
  HTML-элементы и атрибуты, 932
  атрибуты, 405
    доступ к нестандартным HTML-атрибутам, 406
    как узлы типа Attr, 408
    с данными, 407
  вставка и замена элементов документа методами объекта jQuery, 571
  выбор форм и элементов форм, 430
  выбор элементов документа, 393
  выбранные элементы в объекте jQuery, 561
  вычисленные стили, 447
  геометрия и прокрутка, 421
  документа как свойства окна, 380
  документы как деревья элементов, 402
  копирование методами объекта jQuery, 573
  массивов, 164
  методы в библиотеке jQuery для работы с элементами, 972
  обертывание другими элементами с помощью методов объекта jQuery, 574
  определение геометрии, 423
  позиционирование с помощью CSS, 451
  проигрыватель, 987
  размеры, позиции и переполнение, 426
  содержимое, 409
    в виде HTML, 409
    в виде простого текста, 410
    в виде текстовых узлов, 411
  удаление с помощью методов объекта jQuery, 574
  формы HTML, 428
  чтение и запись данных в элементе, 570
  чтение и запись параметров геометрии, 568
  чтение и запись содержимого элемента, 568
элементы-выражения в инициализаторах массивов, 80
эффект тени, 695
  shadowBlur, свойство холста, 695
эффекты проявления и растворения
  fadeIn() и fadeOut(), методы, 586, 588
  fadeTo(), метод, 588

Ю

Юникод, 41
  в идентификаторах, 44
  нормализация, 43
  символы, кодовые пункты и строки JavaScript, 57
  символы управления форматом, 42
  экранированные последовательности, 42

Я

явное преобразование типов, 69
якорные элементы регулярных выражений, 283
  перечень, 284

O'REILLY®

JavaScript. Подробное руководство

Шестое издание книги полностью пересмотрено, добавлены сведения о JavaScript в соответствии с современным положением дел в разработке приложений для Web 2.0. Данное руководство программиста с массой практических примеров может с успехом выполнять функции справочника по базовому языку JavaScript и клиентским прикладным интерфейсам, предоставляемым броузерами. Благодаря добавлению новых глав книга содержит практически все сведения, необходимые для разработки сценариев JavaScript:

- Взаимодействие с протоколом HTTP и использование технологии Ajax
- Обработка XML-документов
- Создание графических изображений на стороне клиента с помощью тега <canvas> и технологии SVG
- Пространства имен в JavaScript, необходимые для разработки сложных программ
- Классы, замыкания, статические значения
- Новые технологии, вошедшие в стандарт HTML5
- Применение библиотеки jQuery

Часть I книги посвящена основам JavaScript. В части II описывается среда разработки сценариев, предоставляемая броузерами. Основное внимание уделяется разработке сценариев с применением методики ненавязчивого JavaScript и модели DOM. Глубокое обсуждение тем клиентского JavaScript иллюстрируется достаточно сложными примерами, которые показывают, как:

- Генерировать оглавление HTML-документа
- Воспроизводить аудио- и видеоролики
- Автоматизировать проверку правильности заполнения форм
- Рисовать динамические круговые диаграммы
- Обеспечить перетаскивание HTML-элементов мышью
- Определять в веб-приложениях горячие комбинации клавиш
- Создавать всплывающие подсказки с применением технологии Ajax
- Применять XPath и XSLT для обработки XML-документов, загруженных средствами Ajax
- Организовать хранение данных на стороне клиента
- Взаимодействовать с сервером, применяя новую технологию WebSockets
- Реализовать параллельное выполнение нескольких заданий с помощью технологии WebWorkers

Часть III – обширный справочник по базовому языку JavaScript с описанием всех классов, объектов, конструкторов, методов, функций, свойств и констант, определенных в JavaScript 1.8, V8 3.0 и ECMAScript 5. Часть IV – справочник по клиентскому JavaScript, в котором описываются API веб-броузеров, стандарт DOM API Level 3 и недавно вошедшие в стандарт HTML5 технологии WebSockets и WebWorkers, объекты localStorage и sessionStorage, а также теги <audio> и <video>.

Категория: веб-программирование/JavaScript

Уровень подготовки читателей: средний

Издательство «Символ-Плюс»
(812) 380-5007, (495) 638-5305

ISBN 978-5-93286-215-5

9 785932 862155

www.symbol.ru